Multicast Communications in Distributed Systems

by

Larry Hughes

PhD Thesis

The University of Newcastle upon Tyne
Computing Laboratory

October 1986

Abstract

One of the numerous results of recent developments in communication networks and distributed systems has been an increased interest in the study of applications and protocols for communications between *multiple*, as opposed to single, entities such as processes and computers. For example, in replicated file storage, a process attempts to store a file on several file servers, rather than one. Multiple entity communications, which allow *one-to-many* and *many-to-one* communications, are known as *multicast communications*.

This thesis examines some of the ways in which the architectures of computer networks and distributed systems can affect the design and development of multicast communication applications and protocols. To assist in this examination, the thesis presents three contributions. First, a set of classification schemes are developed for use in the description and analysis of various multicast communication strategies. Second, a general set of multicast communication primitives are presented, unrelated to any specific network or distributed system, yet efficiently implementable on a variety of networks. Third, the primitives are used to obtain experimental results for a study of intranetwork and internetwork multicast communications.

Acknowledgments

First and foremost, I owe a debt of gratitude to my supervisor, Brian Randell, for many things: from his detailed reading and criticisms of the early drafts of this thesis, to providing me with an environment in which to perform the research, but primarily for suggesting that multicast communications be my research topic. I also wish to thank Santosh Shrivastava who has quietly encouraged my understanding of multicast communications. I am also grateful to Peter Lee who has been a source of encouragement and advice during the closing stages of this thesis and whose suggestions have added both to the contents and the presentation of this thesis.

This thesis could not be what it has become without the advice and contribution of many of the members of the Computing Laboratory at Newcastle. In particular, I must thank Andy Linton for his technical advice during the development of the multicast communication primitives. Many other people have provided assistance, both technical and non-technical, during my stay at Newcastle. They include Graham Parrington, Silvia Pfleger, and Robert Stroud. Finally, the support of my wife, Carolyn and son, Evan, during the past two years should not go unnoticed.

Financial support for my work at Newcastle was provided by a Postgraduate Scholarship from the Natural Sciences and Engineering Research Council of Canada and an Overseas Research Student Award from the Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom.

Table of Contents

List of Figures

Tables

Chapter 1 Introduction

The rapid development of distributed computing systems during the past ten years can be attributed to two forces acting in concert: *technology* (both hardware and software) and *requirements* (both present and future). For example, the dramatic change in the price/performance ratio of computing elements (the technology) spurred the growth and availability of computing power to individuals. This enabled the decentralization of computing power within organizations, resulting in the need to distribute information which had previously been localized. This requirement led, in turn, to further developments in communications technology, partially in Wide Area Networks (WANs), but primarily in Local Area Networks (LANs). These developments are typified, for example, by network transmission speeds: these have increased from thousands of bits per second in the early 1970's to millions of bits per second at the present time.

The interconnection of computers or the like by networks required the definition of **communication protocols**. These specify rules assuring uniform and fair access by machines to networks for the transmission of information. Appropriate protocols were developed to allow individual **processes** on different machines to exchange information. Permitting individual processes to access networks directly, however, demanded further technological novelties, since, for example, applications written for a specific protocol, hence a specific network, could not easily be transferred to machines using different protocols on different networks.

The requirement for portable software and the ability to communicate though multiple networks using different protocols was met by the development of *layered architectures* for distributed systems. Layered architectures were intended, in part, to hide the underlying network and its associated protocol from the communicating processes. By using layered architectures, network specific features, such as packet cyclic redundancy check calculations, can be hidden from the higher layers and features required by an application but not offered by the network, such as file transfer protocols, can be added. Although most distributed systems using layered architectures support a wide variety of protocols (with some layers allowing many different protocols), most, if not all, are intended for *one-to-one* or **unicast communications**. For example, many network protocols allow a transmitting machine to identify at most one receiving machine, while file transfer protocols typically support the transfer of files from a single process to a single file server.

Recently however, applications have been developed which require **multicast communications** or communications between *multiple* processes. For example, in some fault tolerant applications it is necessary to store multiple copies of the same file on separate file servers. Although there are an increasing number of applications which require multicast communications, support for multicast communications is not usually found in distributed systems, or at least, it is not provided for directly. Therefore, multicast communications are often emulated using unicast protocols – making communications to multiple processes a potentially inefficient operation because the same message must be transmitted repeatedly to each possible receiver.

Fortunately, multicast communications are *not* restricted to processes – many local area networks are already capable of transmitting messages from one machine to *all* machines (a **broadcast** transmission) or to *many* machines (a multicast transmission). However, even when implemented on local area networks that actually provide facilities for multicast or broadcast communications, many existing distributed systems do not ordinarily support protocols for multicast communications, at either the process layer or the network layer.

1.1 Thesis Aims

This thesis describes the results of a research project which examined how existing technologies (both hardware and software) could best be used or modified to support multicast communications. The aims of this project were:

a) to describe a set of taxonomies or classification schemes which could be used in the description and analysis of various multicast communication strategies,

b) to develop a general set of multicast communication primitives, unrelated to any specific network or distributed system, yet efficiently implementable on a variety of networks, and

c) to implement and test a multicast communication facility, based upon these primitives, on a variety of network architectures.

1.2 Thesis Contents

The thesis is organized as follows. Chapter Two is divided into four Sections, the first of which introduces some of the terminology associated with multicast communications. The second Section presents a set of three different models for describing communications, to be used in subsequent Chapters for developing multicast communication taxonomies and multicast communication primitives. The third Section consists of a survey of several multicast communication implementations on different networks and distributed systems. In the final Section, a series of multicast applications is discussed, demonstrating some of the potential uses of multicast communications.

Chapter Three presents a series of classification schemes to assist in the understanding and development of intranetwork and internetwork multicast communication systems. The first Section of the Chapter presents a multicast transmission taxonomy employing a simple Transmitter-Receiver paradigm. The second Section develops a multicast response handling taxonomy based upon a Source-Destination model. The third Section discusses the problems associated with identifiers in multicast communications using distributed systems. The Chapter is concluded with a discussion of the possible uses of the taxonomies that have been presented.

Chapter Four describes the design and development of a set of general purpose multicast communication primitives which approach optimal multicast communications, irrespective of the underlying network or internetwork. The primitives themselves are based, in part, upon the needs of the applications described in the second Chapter as well as the taxonomies described in the third Chapter. The Chapter contains four Sections. The primitives required for the management of multicast communications and multicast set membership are discussed in the first Section, while in the second, a series of examples are presented, showing possible uses of the primitives. The third Section compares the proposed primitives with primitives used in existing multicast communication implementations. The Chapter concludes with a review of the primitives and suggests methods of implementation.

Chapter Five describes the implementation and evaluation of the primitives in a distributed UNIX environment using four different network architectures. In the first Section of this Chapter, the facilities (both hardware and software) available in the Computing Laboratory for the implementation and evaluation of the primitives are discussed. The second Section describes the implementation of the primitives using these facilities. In the third Section, performance measurements are presented, comparing multicast communications in the different intranetwork communication environments. The Chapter concludes with some general comments on how the different performance results were obtained and what the results indicate.

Chapter Six examines how different internetwork communication environments can affect multicast communications. The Chapter first presents different methods of identifying Gateways. It then examines the problems associated with identifying Destinations on remote networks. The third Section of the Chapter discusses the design and implementation of several different multicast Gateways in terms of the multicast communication primitives, while performance measurements are presented for three different multicast Gateways in the fourth Section. The final Section reviews the contents of the Chapter, outlining some of the issues that should be considered when performing internetwork multicast communications.

Chapter Seven reviews the aims of the thesis, discusses possible developments of the work presented, and concludes with some final observations.

Chapter 2 Background

This Chapter expands upon the simple description of multicast communications presented in Chapter One through a series of definitions, models, and surveys.

The Chapter is organized as follows. In the first Section, some of the terminology associated with multicast communications is introduced, while in the second Section, methods of describing communications (including multicast) are presented using a series of three different models. A review of many of the known multicast communication implementations is presented in the third Section. In the fourth Section, a series of applications demonstrating some of the uses of multicast communications are discussed. The contents of this Chapter are reviewed in the final Section.

2.1 Terminology

Communications are traditionally discussed in terms of a single transmitting entity (the **Transmitter**) and a single receiving entity (the **Receiver**). For example, the process of communication is frequently described as involving a Transmitter transmitting a **message** to a Receiver, which thus receives a copy of the message. This single Transmitter to single Receiver paradigm, although often satisfactory, represents a special (albeit common) type of communication known as a **unicast** communication.

There are in fact two general cases of communication, both of which are now considered. The first involves a *single* Transmitter transmitting messages to *many* Receivers (where "many" can indicate any number of Receivers, from one Receiver through all possible Receivers) and is commonly known as a **multicast transmission.** There are two "special" classes of multicast transmission, those involving a transmission to a single Receiver (i.e. a unicast transmission) and those involving a transmission to all possible Receivers (known as a **broadcast transmission**).

The second case involves *many* Transmitters (as before, "many" can indicate any number of Transmitters, from one Transmitter through all Transmitters), transmitting messages to a *single* Receiver and is known as a **multicast reception.** The term **unicast reception** is used when describing the reception of a message sent by a single Transmitter. The term **broadcast reception,** indicating that the Receiver receives messages from all Transmitters is rarely, if ever, used.

In both multicast transmission and reception, the set of Receivers (in a multicast transmission) or the set of Transmitters (in a multicast reception) make up the **multicast set.** A multicast set is said to be **static** if the membership is known and does not change over a period of time. Similarly, the membership of the set is considered to be **dynamic** if the membership can change over time.

2.2 Models

Multicast communication, like many other types of communication, can be described using various models which detail different aspects of the communication such as the transmission or reception of messages. In this Section, three such communication models are examined. These models will be used in subsequent Sections and Chapters to assist in the development of several multicast taxonomies and a set of multicast communication primitives.

2.2.1 Transmitter-Receiver Model

As shown above, communications are usually discussed in terms of Transmitters and Receivers. For example, a simple unicast communication was described as involving a single Transmitter sending a **message** to a single Receiver. However, since there may be many *potential* Receivers in any communication, the Transmitter is usually required to identify the intended Receiver using a unique **identifier** (such as a **name** or an **address** [Shoch1978a]) which is transmitted with the message.

However, in a multicast communication, the Transmitter is to transmit a message to a multicast set consisting of one or more Receivers. The *number* of message transmissions performed by the Transmitter depends, in part, upon:

a) how the Receivers are identified, and

b) the number of Receiver identifiers that can be associated with the transmission of the message.

It is possible to define two general types of identifier that can be used to identify the intended Receiver(s) of a message:

a) a **unique** identifier which identifies at most one Receiver, and

b) a **group** identifier which identifies a set of one or more Receivers.

For example, in a multicast communication where there are "R" Receivers, R unique identifiers will be required to represent the members of the multicast set. However, when using group identifiers, anywhere from 1 to R identifiers may be required (1, if all members share the same identifier and R, if the members use group identifiers that are not shared).

The number of identifiers that can be associated with the transmission of a message is either:

a) **one** identifier per message, or

b) **many** identifiers per message.

It is also assumed that with regard to the number of identifiers that can be associated with a message (i.e. one or many), the maximum is fixed (in that neither Transmitter nor any Receiver can vary it), and is known (to the Transmitter and all possible Receivers). In addition, when *many* identifiers are allowed, the number of identifiers that can be associated with a message can vary from one to a maximum of "many".

The Transmitter-Receiver model is useful in that it can describe the (ideal) number of messages that must be generated to ensure that all members of the multicast set can receive a copy of the message. For example, if Receivers can only be identified with unique identifiers and only one identifier can be supplied with each message, the Transmitter will be forced to send as many messages as there are members of the multicast set. However, if all the intended Receivers share a common group identifier, the Transmitter need only send one message.

2.2.2 Source-Destination Model

In certain applications, a Transmitter may expect to receive messages from one or more Receivers. For example, a Transmitter may request the current time of day by transmitting a "time request" message. Upon receipt of the "time request" message, a Receiver could transmit its time of day to the original Transmitter.

To avoid the confusion of describing a Transmitter (or a Receiver) as *both* a Transmitter and a Receiver, network designers have developed models in which the communicating entities have the properties of both Transmitters and Receivers. Such models include the **Source-Destination** model, the **Client-Server** model and the **Contractor-Bidder** model to name but three.

For example, in the Source-Destination model, the **Source** is first a Transmitter and then (possibly) a Receiver, whilst the **Destination** is first a Receiver, then (possibly) a Transmitter:

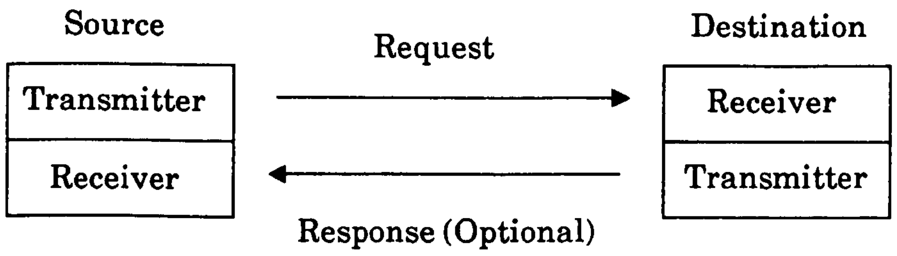

Figure 2-1: The Source-Destination Model

In a typical unicast communication, the Source transmits a message (the **Request**) to a Destination which is acting as a Receiver. Depending upon the application, the Destination may be expected to respond with another message (the **Response**) to the Source, now acting as a Receiver.

However, in a multicast communication, the number of Responses expected by the Source can vary from none (in which no Responses are anticipated) to as many as there are Destinations (if all Destinations are expected to respond). Depending upon the application, the number and identity of the Destinations responding may be important. For example, a Source may request the members of a multicast set to respond once they have completed a certain task. If the membership of the multicast set is known to the Source (for example, using a static multicast set), determining which members have not responded is a simple matter. However, if the membership of the multicast set is dynamic, identifying the Destinations which have not responded is clearly a more difficult operation, requiring, for example, the use of a protocol.

The Source-Destination model can therefore be used in comparing different multicast response handling implementations given that the method of identifying the members of the multicast set is known.

2.2.3 Layered Architectures

The need for and uses of layered architectures were outlined in Chapter One. However, there is not a single, universal layered architecture model (contrary to what some network designers would have us believe [Cohen1983a]), instead, various layered models exist, supporting a wide range of layers.

Although the models themselves may differ, the individual layers within any architecture exhibit similar characteristics. For example, each layer has its own protocol and types of identifier. Any communications between the entities that make up a layer (known as the (N)-Layer) take place using the **communication services** offered by the next lower layer (the (N-1)-Layer)). The following diagram illustrates the interaction between any two adjacent layers (the (N)-Layer and the (N-1)-Layer) [Bochman1985a]:

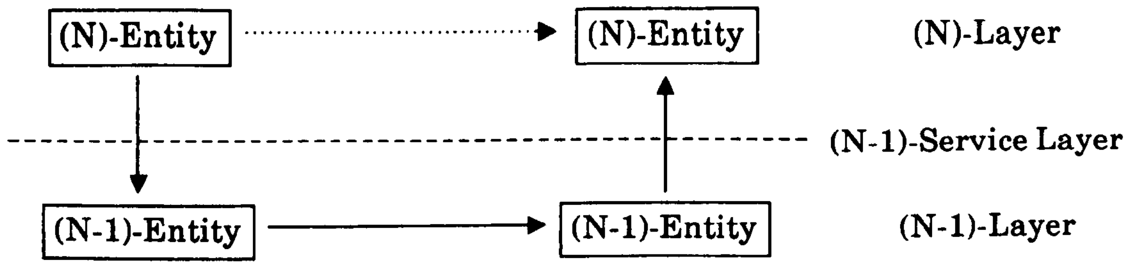

Figure 2-2: Interaction of Adjacent Layers

As an example, consider the following three layer model (based upon the Arpanet [Padlipsky1985a]) which can be used to convey the basic concepts of a layered architecture:

Process (or **Application**) **Layer**: consisting of Processes which can receive transmissions from other Processes (on this or other Hosts) or can perform transmissions to Processes (once again, on this or other Hosts). Hosts usually support many Processes and distinguish between them using some type of Process identifier;

Communication Layer: the Communication Layer hides the idiosyncrasies of the various networks to which the Host may be connected and attempts to offer a uniform view of the different networks to the Process(es) requiring the use of Processes on other Hosts. There is normally only one Communication Layer on each Host and it is responsible for directing messages between the Network and Process Layers;

Network Layer: is responsible for the transmission and reception of messages on a specific network. A Host can be connected to many networks.

In a layered architecture, it is possible that an identifier is expanded into several more identifiers [Watson1983a]. For example, in the Arpanet model described above, a transmitting Process could send a message to a receiving Process using the lower layers to perform the transmission. In this case, the identifier supplied by the Process would be required to map into at least two additional identifiers: the first identifies the receiving **Host** (to allow the message to be sent across a network to the correct Host) and the second identifies the receiving **Process** (to allow the receiving Communication Layer to supply the message to the intended Process).

It is important to note however, that in most layered architecture models, the receiving Process is *not* explicitly identified as suggested above. Instead, the transmitter of the message often identifies a destination **Port**, rather than a specific Process – since it is assumed that a Port number can be kept static, whereas Process identifiers are usually dynamic. In a layered architecture such as the one described above, Processes are often identified using a **Host-Port** identifier.

Layered architectures should be considered when dealing with multicast communications. For example, the number of messages generated by any one layer in support of a multicast transmission depends, in part, upon how the layer's receiving entities are identified. Similarly, in a multicast reception, the (N)-Layer Receiver may require the (N-1)-Layer Receiver to queue messages until they can be processed, potentially affecting the performance of the machine.

2.3 Survey of Existing Systems

In the following Section, several existing networks and distributed systems are discussed in light of how they support multicast communications. This survey is presented in three parts. First, those implementations designed for the Ethernet, a local area network which can support an efficient form of multicast communications, are considered. Second, multicast communication implementations intended for the Cambridge Ring, a local area network not directly supporting an efficient form of multicast communications, are examined. Finally, a series of multicast communications implementations primarily intended for specific research networks other than the Ethernet or the Cambridge Ring are presented.

2.3.1 The Ethernet [Metcalfe1975a, Hopper1986a, Chorafas1984a]

The Ethernet is a bus-structured local area network, developed by Xerox, allowing up to 1024 stations on a single network. Each Ethernet station can be associated with its own unique identifier (allowing unicast communications), a group identifier shared by *all* stations (permitting broadcast communications) and, in some implementations, several group identifiers which can be shared by any number of stations [DEUNA1983a] (thus making efficient multicast communications possible).

Messages are transmitted across the Ethernet in **packets**. The format of the packet depends upon the version of the Ethernet in use. The Xerox Ethernet packet format is as follows [DIX1980a]:

Preamble	Destination Address	Source Address	Type	Data	Frame Check Sequence
8 bytes	6 bytes	6 bytes	2 bytes	46 to 1500 bytes	4 bytes

Figure 2-3: Xerox Ethernet Packet Structure

However, the IEEE 802.3 standard [IEEE1982a] replaces the **Type** field (used to indicate the type of message, thereby allowing a number of different protocols to be supported) with a **Length** field (indicating the number of bytes in the **Data** field).

Since the Ethernet is a bus structured network, all stations on the network receive a copy of the transmitted packet. However, it is the responsibility of each station to determine whether the message should be kept or ignored by examining the Destination address.

Because of the Ethernet's ability to support both unique and group identifiers many network designers have utilized the Ethernet in the development of multicast communication protocols as the following examples demonstrate.

2.3.1.1 Pup [Boggs1983a]

PUP is a packet structure designed to be transmitted in an Ethernet packet. While the Destination address with the Ethernet packet only specifies a Host, the PUP address specifies both a Host and a Port. For a Process to receive a PUP packet, it must first be associated with a Port; thereafter, any packets arriving at the Host with a Port number the same as that of the Process will be made available to the Process.

PUP supports a form of multicast transmission by allowing the Source Process to broadcast a message with a Port number which is common to Processes on different Hosts on the Network. It is the responsibility of each Destination Host to filter the incoming packet using the PUP Port number to determine if the requested Port is available.

2.3.1.2 The V-System [Cheriton1984a, Cheriton1985b]

The V-System is a distributed kernel developed at Stanford University which uses an Ethernet as an "extended backplane" to connect a group of diskless SUN workstations and server machines.

The V-System utilizes the underlying Ethernet broadcast address and its own packet structure to transmit multicast messages to servers belonging to multicast sets. As in the case of PUP, it is the responsibility of each Destination Host to determine whether it supports the requested service by examining the group identifier supplied with each message.

The need to support multicast reception has been recognized by the designers of the V-System in that a Process can request the kernel to discard all incoming messages, return the first message received (discarding the rest) or return all messages (individually, as they are received).

2.3.1.3 MP [Ahamad1985a]

MP is a multicast transmission implementation developed at SUNY, Stony Brook which uses an Ethernet to connect a series of UNIX Hosts. The implementation has extended UNIX 4.2 sockets to allow a message to be sent from a Source Process to one or more Destination Processes. As in other

Ethernet implementations, multicast transmissions are achieved using broadcast transmissions.

2.3.2 Cambridge Ring [JNTIS1982a, JNTPS1982a, Needham1982a]

The Cambridge Ring is a slotted-ring local area network consisting of up to 254 stations (or nodes). Each node on the network is associated with its own unique address (allowing unicast communications). For a communication to take place, the Source Cambridge Ring node first waits for a free "slot" to become available; it then supplies the network with the address of a Destination Cambridge Ring node and two bytes of data in a **mini-packet**. When the mini-packet arrives at the Destination, a copy of the data is taken and a pair of Response bits (within the mini-packet) are set to indicate whether the mini-packet was accepted or rejected. The mini-packet then returns to the Source node which reads the Response bits and frees the mini-packet (making it a free slot once again).

Because the Cambridge Ring was intended for unicast communications, there have been few multicast communication implementations for the Cambridge Ring described in the literature. However, one implementation using the Cambridge Ring which does support a form of multicast communications is the UNIVERSE project [Leslie1984a, Waters1984a], which links several Cambridge Rings by a satellite broadcast channel. Network Layer multicast communication is achieved by the use of protocols which allow the receiving satellite stations to determine whether the intended Destinations reside on the local Cambridge Ring before transmitting the message on the local ring.

2.3.3 Other Implementations

Metanet [Aguilar1984a] is an extension of the United States' Department of Defense Internet Protocol (IP) intended, in part, to allow multicast communications. By utilizing some of the spare space within the header of an Internet Protocol packet, Metanet can transmit an additional eight destination addresses. How these additional addresses are used depends upon the underlying network address structure. For example, in a broadcast network, each Destination Host receiving a copy of the packet could examine the eight addresses to determine if its own address was included in the list.

The Admiralty Surface Weapons Establishment (ASWE) **Serial Data Highway** [Lakin1982a], is a local area network developed to provide fault tolerant communications between distributed computers for ship borne command and control systems. The highway connects up to 63 devices (known as terminals), each of which may communicate with any other, and a Poller, which controls the traffic between the terminals. Terminals do not transmit unless they have been polled by the Poller. Once polled, the terminal responds with one of: a null message, a message for a specific terminal, or a multicast message for a group of terminals. Terminals receiving the broadcast message act a Servers, returning responses when they are polled.

Shoshin [Tokuda1983a] is an experimental system designed at the University of Waterloo to study the development and evaluation of distributed software. The Shoshin system consists of two PDP-11/45s and ten LSI-11/23s, all connected by a high speed parallel bus. Shoshin has a layered architecture with Source applications sending requests to one, many, or all possible Destinations.

The **IBM Token Ring** [Janson1983a] supports both intranetwork multicast communications (i.e. multicast communications within a single network) and internetwork multicast communications (i.e. multicast communications across several networks) using a hierarchical identifier structure. The identifier consists essentially of two parts, a ring identifier (which can identify the current ring, a specific ring, or all rings) and a station identifier (which can indicate a specific station, all stations, or a group of stations). Depending upon the identifier structure chosen, unicast, multicast, or broadcast communications can be achieved between a transmitting station and receiving station(s) on a specific ring or all rings.

MIKE (Multicomputer Integrator KErnel) [Tsay1983a] is a distributed network developed at the Ohio State University. The network connects a series of LSI-11/23s and a PDP-10 using a double-loop ring. Although the main emphasis of the research on MIKE is the development of guardians for distributed processors, the underlying network does permit multidestination transmissions.

Hubnet [Lee1983a] is a network developed at the University of Toronto based upon fibre-optic links. Both transmitters and receivers are interconnected via a "hub" consisting of two parts: a **selection** hub and a **broadcast** hub. A transmission involves a transmitting Host sending a message to its selection hub, which in turn transmits the message to all Hosts (and possibly other selection hubs) attached to the broadcast hub (including the Host which originally transmitted the message – thereby allowing the Host to determine whether the hub accepted or ignored the message). Since each message transmitted by the hub is sent to all Hosts (i.e. a hub transmission is a Network Layer broadcast transmission), Hubnet has the potential of supporting multicast communications.

2.4 Applications

In the previous Section, several examples of the technology available for the support of multicast communications were discussed. In this Section, four applications are presented, each of which illustrate some of the different requirements that a multicast communication facility could be expected to support.

2.4.1 A Time-Signal Generator

The simplest type of multicast communication is one in which the Source transmits a message to the members of the multicast set and none of the Destinations are to respond. In the following example, the Source is a "time-signal generator", that is, a Process that continuously supplies the time to the members of a multicast set. A Destination in this example is any Process that happens to need the time. The time value is obtained by "tuning into" the multicast set associated with the time-signal generator.

The Source Process (i.e. the time-signal generator) algorithm is as follows:

```
repeat
    GetTime;
    SetupTimeSignalPacket;
    SendTimeSignalToTimeReceiverSet;
until
NetworkOrClockFails;
```

(It is assumed that before each call to GetTime, there is a brief pause which stops the Source Process from flooding the network with time values that hardly differ from each other.)

For a Process to obtain a time value, it must first become a member of the multicast set to which time signals are sent. The Destination Process (i.e. one requiring the time) must then wait for a message containing the time value (the "Time Signal") to arrive. Should the Destination miss the *current* time-signal, it is assumed that there will be another sent by the time-signal generator after a brief interval (since new time-signals are continually being transmitted). The Destination Process algorithm is as follows:

```
    WaitForATimeSignal;
    SetupTimeFromTimeSignal;
```

This code fragment takes the first "TimeSignal" that is received and uses it to establish the Process's current time.

Although the above algorithms are intended to allow a Process to receive a time value from a time-signal generator, the accuracy of the received time value can depend upon how the Destinations are identified. For example, should the underlying network use unique identifiers, significant delays could occur during the repeated transmission of a time-signal (with the same time value), rendering the received value very inaccurate. Boggs [Boggs1983a] has suggested that time servers (i.e. the time-signal generator) send an error value with each (broadcast) message, thereby allowing the Process requiring the time to choose the most accurate time value possible. However, if the multicast transmission is simulated by a series of unicast transmissions, the concept of a single error value is meaningless since the error increases with each message sent.

2.4.2 A Time Server

In this Section, another time-server application is presented. However, unlike the previous example, the Source is the Process *requesting* the time and the Destinations are Processes which *return* their current time value as Responses.

The Source Process (i.e. the Process requesting the time) could be written as follows:

```
SendTimeRequestMessageToTimeServers;
ReceiveFirstAvailableTimeMessage;
SetupTimeFromTimeMessage;
```

In the above algorithm, the Source Process accepts the first available time-message, ignoring any subsequent time-messages.

However, there are situations where the time value accepted from the first received time-message may not be sufficiently accurate for the Source Process. For example, a user may be satisfied obtaining a time value accurate to the nearest minute, whereas some applications may require a time value accurate to within a few milliseconds. To overcome this problem, the incoming time-messages could be *filtered.*

Filtering can be performed either by the Source Process directly or by a time-message reception procedure supplied by the Source. In this and subsequent multicast application examples, filtering will be performed by a supplied function, thereby freeing the Source of having to explicitly program items such as exception handling.

For example, a Source Process could be supplied with a function which returns a success indication if a time-message was received satisfying the required accuracy (in this algorithm, the ReceiveTimeMessage function accepts a value indicating the number of milliseconds of accuracy required):

```
SendTimeRequestMessageToTimeServers;
if ReceiveTimeMessage(FiftyMilliseconds) = Success then
    SetupTimeFromTimeMessage
else
    error("No sufficiently accurate time value found.");
```

The Destination Process (i.e. the Process supplying the time-message) could be written as follows:

```
repeat
    WaitForATimeRequestFrom(Source);
    GetTime;
    SetupTimeMessage;
    SendTimeMessageTo(Source);
until
NetworkOrClockFails;
```

Note, in this algorithm, the Destination Process maintains a variable "Source" which identifies the originator of the Request. This allows the Destination Process to return the time-message to the requesting Source Process.

2.4.3 A Triple-Modular Redundant File-Server

A Triple-Modular Redundant (or TMR) File-Server consists of three Destinations (the **file-servers**) responsible for storing separate copies of a file sent by a Source. The file transfer continues until either the file has been successfully received by the file-servers or the number of file-servers falls below two (that is, the file transfer will continue with two *or* three file-servers). In this example, it is assumed that for the duration of the file transfer there is only a single Source Process (i.e. other Processes requiring the use of the file-server must wait until any file transfers in progress have completed) and lost messages are not retransmitted.

In the following algorithm, the rules governing the transmission of messages by the Source Process are as follows:

- a Destination must respond within a certain time period otherwise it is assumed to be inactive;
- if a Destination is found to be inactive, it should be ignored (i.e. not transmitted to and have any subsequent Responses discarded);

- if two or more Destinations are found to be inactive (that is, Responses are not received from them within a certain time period), the transmission is to be stopped.

These rules are enforced by the function ReceiveAndFilterReplies, which returns a status value to the Source Process of either "TMRContinue" if the transmission can continue or "TMRFailed" if the transmission is to be aborted.

The Source Process (i.e. the Process requiring the services of the TMR File-Server) algorithm could be written as follows:

```
GetFileHeader;
SendToTMRFileService(FileHeader);
status := ReceiveAndFilterReplies;
if status < > TMRFailed then
repeat
      GetFileBlock;
      SendToTMRFileService(FileBlock);
      status := ReceiveAndFilterReplies;
until
(status = TMRFailed) or (FileBlock = EndOfFile);
```

The Source Process first sends the file header and then a series of file blocks, the last of which contains an end of file indicator. After each transmission, the Source Process waits for the function ReceiveAndFilterReplies to respond with a status value, indicating whether to continue the file transfer. The ReceiveAndFilterReplies function collects Responses from those Destinations still involved in the transfer and which respond within a fixed time period. Should the file transfer fail, the Source Process simply stops transmitting. (It is assumed that the Destinations will also eventually "time-out" and abort the file transfer if messages are not received from the Source Process within a corresponding time period.)

The Destination Process (i.e. the File-Server) algorithm could be written as follows:

```
repeat
    WaitForFileHeader;
    SetupFileHeader;
    ReplyDoneMessage;
    repeat
        status := WaitForFileBlock;
        if status = Okay then
        begin
            PutFileBlock;
            ReplyDoneMessage;
        end;
    until
    (FileBlock = EndOfFile) or (status < > Okay);
    if status < > Okay then
        AbortFileCreation;
until
HostOrNetworkFailure;
```

The File-Server creates a new file from the supplied file header and then proceeds to write each received file block to the newly created file. The File-Server uses two different receive functions. The first, WaitForFileHeader, waits indefinitely for a file header. Once the transmission has started, a second receive function, WaitForFileBlock, expects a File Block to be made available within a certain time period - this is to ensure that the File-Server does not wait indefinitely for a slow or inactive Source. File Blocks received within this period are returned with a status of "Okay", otherwise an error is returned and the creation of the file is aborted.

The TMR File-Server example demonstrates the need for an "intelligent" multicast filter, which, unlike the filter presented in Section 2.4.2, must maintain state information, allowing it to determine which Responses are valid and which are to be ignored. In addition, it demonstrates the simplicity of a multicast transmission in that the Source need only send one message which is to be received by the members of the multicast set, rather than repeatedly transmitting the same message to each member.

2.4.4 A Two-Phase Commit Protocol

A Two-Phase Commit protocol attempts to ensure that a transaction, such as a database update, is either executed entirely or not at all by the Destinations (or **Participants**) offering the service. Briefly, the first phase consists of a Source (the **Commit Coordinator**) asking all Participants whether they can commit themselves to performing a specific (common) action. Those that can indicate by a "yes" vote and those that cannot respond with a "no" vote. Should one or more Participants indicate that they cannot commit themselves to perform the required action, the Commit Coordinator transmits an "Abort Request", to which all Participants must also respond.

However, if *all* Participants indicate "yes", the Commit Coordinator then issues a "Commit Request", in which the Participants guarantee the performance of the required action - this is the second phase of the protocol.

The following algorithms, which are adapted from [Bennett1984a], describe a Two-Phase Commit protocol. The Commit Coordinator (the Source) algorithm is as follows:

```
{ First Phase }
SendCanYouCommitMessage;
answer := WaitForResponses;
if answer = Yes then
begin
    { Second Phase }
    set(COMMIT);
    repeat
        SendMessage(DoCommit);
        answer := WaitForResponses;
    until
    answer = Yes;
    reset(COMMIT);
end
else
begin
    repeat
        SendMessage(DoNotCommit);
        answer := WaitForResponses;
    until
    answer = Yes;
end;
```

The WaitForResponses procedure used by the Commit Coordinator returns either a "yes" or "no" answer, depending upon the number of Responses (and their values) received within a certain time period. A "yes" indicates that all possible Destinations responded with a "yes" vote, while a "no" indicates that one or more Participants could not ensure completion of the Two-Phase Commit.

Once the decision is taken to either commit or abort the transaction, the Commit Coordinator continues to transmit "DoCommit" messages (if the commit is to take place) or "DoNotCommit" messages (if the commit is not to take place) until all Participants have responded with a "yes" indication. (The statements "set(COMMIT)" and "reset(COMMIT)" are specific to the Two-Phase Commit protocol and are not considered here.)

A Participant (i.e. a Destination) has the following algorithm:

```
WaitForACanYouCommitMessage;
EnsureUndoRedo;
if CannotEnsure then
      SendMessage(NoVote);
else
      SendMessage(YesVote);
WaitForVerdict;
if Verdict = DoCommit then
begin
      DoUpdate;
      SendMessage(YesVote);
end
else
begin
      UndoUpdate;
      SendMessage(NoVote);
end;
```

The Participant initially waits (indefinitely) for a commit request. When a commit request arrives, the Participant attempts to ensure that it can actually commit itself. Once the decision is taken (to either commit or abort), the Participant must wait for the Commit Coordinator to indicate the verdict. (The procedures "EnsureUndoRedo", "DoUpdate" and "UndoUpdate" as well as the Boolean variable "CannotEnsure" are all specific to the Two-Phase Commit protocol.)

The Two-Phase Commit protocol described in this Section has shown how multicast communications can assist in the implementation of atomic events. That is, the multicast transmission is treated as a single event which is either completed or ignored, rather than burdening the application with the details of the success or failure of individual communications.

2.5 Concluding Remarks

In this Chapter, some of the basic concepts associated with multicast communications such as the terminology and ways of describing multicast communication implementations have been covered. In addition, a survey of many of the known multicast communication implementations was presented, demonstrating that both hardware and software facilities do exist which can support multicast communications. Finally, a set of applications were discussed, showing that there are many different requirements which must be considered in a multicast communication.

In the next two Chapters, several of the topics covered in this Chapter will be expanded to further our understanding of multicast communications. In Chapter Three, the communication models will be used to develop a series of taxonomies which will, in turn, be used to describe the implementations and applications discussed in this Chapter. In addition, in Chapter Four, the following common features, described in Section 2.4, will be used to to develop a set of multicast communication primitives comprising of:

a) the ability to *send* a message to the members of multicast set;

b) the ability to *receive* multiple messages sent by one of more transmitters, and to filter those messages not required;

c) the ability to *join* (or become a member) of a multicast set and receive messages which are destined to the members of the set;

d) the ability to *leave* a multicast set, thereby ignoring any subsequent messages sent to the members of the set.

Chapter 3 Methods of Classifying Multicast Communications

In Chapter Two, three different models of describing a multicast communication were presented: a Transmitter-Receiver model, a Source-Destination model, and a Layered Architecture model. In this Chapter, these models are used to develop a set of classification schemes for describing different aspects of multicast communications. These classification schemes will be used in subsequent Chapters for the development, implementation and testing of a set of multicast communication primitives.

A multicast transmission taxonomy is presented first, based upon the Transmitter-Receiver model. A multicast reception taxonomy is then developed using the Source-Destination model. Finally, the Layered Architecture model is examined and used to describe both multicast transmission and the reception of multicast messages in some types of distributed systems.

3.1 A Multicast Transmission Taxonomy

The Transmitter-Receiver communications model presented in Chapter Two describes two factors which can influence any communication – the type of identifier used to identify the Receiver (**Unique** or **Group**) and the number of identifiers that can be associated with a single message (**One** or **Many**). By enumerating these two factors, a multicast transmission taxonomy can be developed and used to describe several different aspects of multicast communications.

3.1.1 The Basic Taxonomy

The multicast transmission taxonomy developed in this Section is intended to establish an ideal lower and upper bound for the number of transmissions required by a Transmitter in order that all members of the multicast set receive a copy of the message. (Note, the present analysis does *not* take into account acknowledgments or message retransmission.) The taxonomy itself is made up of four different classifications based upon the number of identifiers that can be associated with a message (One or Many) and the type of identifier used to identify the Receivers (Unique or Group). Table 3-1 (below) lists the different classifications and their associated minimum and maximum number of transmissions:

Classification	Number of Transmissions	
	Minimum	Maximum
One-Unique	R	R
One-Group	1	R
Many-Unique	1	S
Many-Group	1	S

Table 3-1: The Basic Taxonomy

where:

R - is the total number of Receivers in the multicast set, and

S - is the ceiling of **R** / **N**, where **N** is the number of identifiers (i.e. "many") which can be associated with the transmission of a single message.

There now follows a discussion of each of the classifications in terms of the minimum and maximum number of transmissions required:

One-Unique: a Transmitter in the One-Unique category can associate at most one identifier with each message, while the Receivers are represented using unique identifiers. In this category, the minimum and maximum number of transmissions are the same, that is **R**, since each Receiver must be transmitted to individually;

One-Group: a Transmitter in the One-Group category can associate at most one identifier with each message, while the Receivers are represented using group identifiers. The number of transmissions depends upon the number of group identifiers required to represent the members of the multicast set – if all members share the same group identifier, then only one transmission is required. However, in the worst case, where each Receiver is identified using its own group identifier, a total of **R** transmissions are required;

Many-Unique: a Transmitter in the Many-Unique category can associate many identifiers with each message, while the Receivers are represented using unique identifiers. The number of transmissions depends upon the number of multicast set members (**R**), and the number of identifiers that can be associated with the message (**N**):

$R < N$: resulting in a single transmission;
$R = N$: resulting in a single transmission;
$R > N$: resulting in multiple transmissions, to a maximum of **S**.

Many-Group: a Transmitter in the Many-Group category can associate many identifiers with each message, while the Receivers are represented using group identifiers. The number of transmissions depends upon the number of group identifiers required to represent the members of the multicast set **G** (with a value between 1 and **R**), and the number of identifiers that can be associated with the message (**N**):

$G < N$: resulting in a single transmission;
$G = N$: resulting in a single transmission;
$G > N$: resulting in multiple transmissions, to a maximum of **S** (when each Receiver is identified with its own group identifier, i.e. $G = R$).

Finally, it is worth noting that some systems may support hybrid schemes whereby a multicast set is represented by both types of identifier (i.e. Unique and Group). In these situations, the taxonomy still can be used, however all identifiers should be considered as *group* identifiers (since a Group identifier is assumed to represent one or more Receivers).

The following examples demonstrate how the multicast transmission taxonomy can be used to classify different networks and distributed systems.

3.1.1.1 The Cambridge Ring [Needham1982a]

A transmission on the Cambridge Ring involves a Source station (the Transmitter) supplying two bytes of data and the address of a single Destination station (the Receiver) in a mini-packet to the Cambridge Ring. The mini-packet travels around the ring, past the intended Destination (which takes a copy of the mini-packet and sets the Response bits) and back to the Source (which checks the Response bits and frees the mini-packet).

From this description of a Cambridge Ring transmission, the Cambridge Ring can be classified as a One-Unique multicast transmission network. Therefore, in a multicast transmission involving "R" Cambridge Ring stations, a total of R transmissions would be required.

Note, the Cambridge Ring can also support One-Group multicast transmission by performing a *broadcast* transmission to all stations. However, since Cambridge Ring broadcast transmissions are rarely discussed in the literature (in part because all stations access the same pair of Response bits, potentially destroying the previous value [JNTIS1982a]), One-Group multicast transmission on the Cambridge Ring is not considered in detail here.

3.1.1.2 The Ethernet [Metcalfe1975a]

On an Ethernet, messages are transmitted in packets containing a single Receiver identifier. However, because of the bus structure of the Ethernet, the packets are potentially available to all stations on the network. The identifier itself can be either Unique, identifying a single receiving station, or Group, identifying a group of receiving stations. Depending upon the type of receiving hardware available, the Group identifier can either be recognized by all possible Receivers or a subset of them (for example, see [DEUNA1983a]).

Clearly the Ethernet can be classified using the multicast transmission taxonomy as either a One-Unique or a One-Group multicast transmission network. For example, it is One-Unique if all stations belonging to a multicast set can only be identified by their Unique station identifier - causing the transmitting Ethernet station transmit a copy of the message to each station in turn. However, it is One-Group if all stations share the same Group identifier, minimizing the number of messages the transmitting Ethernet station must send.

3.1.1.3 Metanet [Aguilar1984a]

A Metanet station is designed to support both the transmission and reception of messages associated with up to nine identifiers (either Unique or Group, depending upon the station identification scheme of the Destination network). Therefore, depending upon the type of identifier the Destination Metanet stations are represented by, Metanet can be classified as either a Many-Unique or a Many-Group multicast transmission network.

3.1.1.4 The V-System [Cheriton1984a]

A V-System Client Process (i.e. a Transmitter) on one machine sends a message to a multicast set consisting of one or more Server Processes (i.e.

Receivers), all sharing a common "Group Identifier". Interprocess multicast transmission in the V-System can therefore be classified as One-Group. In addition, since the Client uses a single Group Identifier to identify all possible Servers in the multicast set, the V-System guarantees a minimum number of transmissions by the Client.

3.1.1.5 Summary

In this Section a taxonomy has been developed which allows the description of multicast communications between a Transmitter and a multicast set consisting of one or more Receivers. The taxonomy is based upon the number of identifiers (either One or Many) and the type of identifier (either Unique or Group) which can be associated with the transmission of a message.

However, the taxonomy as presented does not describe the effect that layered architectures can have upon multicast transmissions, nor does it allow an examination of the overheads involved in message distribution. These features will be discussed at length in the next Section.

3.1.2 The Multicast Transmission Taxonomy and Layered Architectures

Since distributed systems may consist of a variety of networks, it is often desirable to hide the underlying network and its associated protocols from application Processes, thus permitting common software to be run on different machines on different networks. In order to allow the portability of software and to allow entities on different machines to communicate, most distributed systems are designed in a layered fashion [Watson1983a].

In Chapter Two, it was shown that although communications in a layered system between two (N)-Entities may appear to occur at the (N)-Layer, the communication is in fact using the communication services of the next lower layer, the (N-1)-Layer. It is the (N-1)-Layer that dictates the number and type of identifiers which can be used by the (N)-Layer.

The following assumptions are made regarding layered architectures:

a) the transmitting (N)-Entity supplies the transmitting (N-1)-Entity with a message, and the number and type of identifiers which the (N-1)-Service Layer allows. For example, if the (N-1)-Service Layer supports Many-Unique transmission (that is, the transmitting (N)-Entity can supply many unique identifiers with each message), then each message is supplied to the transmitting (N-1)-Entity with many unique identifiers;

b) when the transmitting (N-1)-Entity receives a message, plus identifier(s), from the (N)-Entity, the message is transmitted immediately without waiting for additional messages or identifiers. For example, if the (N)-Layer supports One-Unique transmission and the (N-1)-Layer supports Many-Unique transmission, each message supplied by the transmitting (N)-Entity is transmitted with a single unique (N-1)-Identifier by the transmitting (N-1)-Entity (even though the (N-1)-Message could have been associated with more (N-1)-Identifiers);

c) the number of receiving (N-1)-Entities is always less than or equal to the number of receiving (N)-Entities. That is, a receiving (N)-Entity can be associated with at most one (N-1)-Entity, whereas a receiving (N-1)-Entity can be associated with any number of (N)-Entities.

The amount of processing required by the system to support the communication depends, in part, upon how the individual layers are classified by the multicast transmission taxonomy. For example, if a single (N)-Identifier is used to identify all receiving (N)-Entities (that is, the (N)-Layer supports One-Group multicast transmission) but the transmitting (N-1)-Entity must use separate (N-1)-Identifiers to identify each receiving (N-1)-Entity (that is, the (N-1)-Layer only supports One-Unique multicast transmission), then the One-Group (N)-Identifier must be mapped into a series of One-Unique (N-1)-Identifiers (by either the transmitting (N)-Entity or the transmitting (N-1)-Entity). Similarly, little or no processing may be required if both the transmitting (N)-Entity and the transmitting (N-1)-Entity support the same number and type of identifier.

Clearly, additional processing may be required when either or both of the following situations arise:

a) the (N)-Identifier type and the (N-1)-Identifier type differ. For example, the (N)-Identifier may be group (unique) and the (N-1)-Identifier may be unique (group);

b) the number of (N)-Identifiers required to identify the receiving (N)-Entities differs from the number of (N-1)-Identifiers required to identify the receiving (N-1)-Entities. For example, one (many) (N)-Identifiers are used at the (N)-Layer while many (one) (N-1)-Identifiers are required by the (N-1)-Layer. Similarly, the layers may support different numbers of "many" identifiers.

In both of these circumstances, it may be necessary to change one or both of:

a) the identifier types - from group to unique or unique to group;

b) the number of identifiers - from one to many, many to one, or many to many.

The processing requirements of the adjacent layers within a system can be expressed in a tabular form, based upon how the layers are classified by the multicast transmission taxonomy developed in the previous Section:

Classification of (N-1)-Layer	Classification of (N)-Layer			
	One		Many	
	Unique	Group	Unique	Group
One-Unique	U → U 1 : 1	G → U 1 : 1	U → U m : 1	G → U m : 1
One-Group	U → G 1 : 1	G → G 1 : 1	U → G m : 1	G → G m : 1
Many-Unique	U → U 1 : n	G → U 1 : n	U → U m : n	G → U m : n
Many-Group	U → G 1 : n	G → G 1 : n	U → G m : n	G → G m : n

Table 3-2: Processing Requirements of Adjacent Layers

Each entry in Table 3-2 describes the changes associated with the (N)-Identifier(s), in terms of identifier type and number of identifiers, if they are to be mapped into (N-1)-Identifiers.

Identifier translation, when required, is represented by the arrow (→). The four possible identifier translation combinations should be read as follows:

U → U - each unique (N)-Identifier supplied with the message is mapped into a unique (N-1)-Identifier;

U → G - each unique (N)-Identifier supplied with the message is mapped into a group (N-1)-Identifier;

G → U - each group (N)-Identifier supplied with the message is mapped into one (or more) unique (N-1)-Identifiers;

G → G - each group (N)-Identifier supplied with the message is mapped into one (or more) group (N-1)-Identifiers.

The change in the number of identifiers (that is, the difference between the number required to identify the receiving (N)-Entities and the number required to identify the receiving (N-1)-Entities) associated with each message is represented by the colon (:) and is in the form **m : n**, where **m** indicates the maximum number of (N)-Identifiers allowed with each message by the (N)-Layer, and **n** is the maximum number of (N-1)-Identifiers allowed with a single message by the (N-1)-Layer. The possible identifier changes and their impacts are:

1 : 1 - One (N)-Identifier per message to One (N-1)-Identifier per message. Both the (N)-Layer and the (N-1)-Layer support at most one identifier with each message, irrespective of the number of (N-1)-Identifiers produced by any mapping. For example, if the identifier type changes from Group to Unique (G → U) or group to group (G → G), each (N-1)-Identifier produced must be sent with its own copy of the message, even though only one message is supplied by the (N)-Entity. This can result in many (repeated) transmissions of the same message to the receiving (N-1)-Entities.

1 : n - One (N)-Identifier per message to Many (N-1)-Identifiers per message. The (N)-Layer allows at most one (N)-Identifier per message, but the (N-1)-Layer allows many (N-1)-Identifiers with each message. When the one (N)-Identifier maps into several (N-1)-Identifiers (such as Group to Unique (G → U) or possibly Group to Group (G → G)), then the number of repeated transmissions of the message by the transmitting (N-1)-Entity can be less than the 1 : 1 case (above).

m : 1 - Many (N)-Identifiers per message to One (N-1)-Identifier per message. The (N)-Layer allows many (N)-Identifiers with each message, but the (N-1)-Layer allows at most one (N-1)-Identifier with each message. If the (N)-Identifiers are Unique, the maximum number of messages sent by the transmitting (N-1)-Entity will equal the total number of (N)-Identifiers supplied by the (N)-Entity if Unique to Unique (U → U), but the total can be less if several Unique identifiers map into a single Group identifier (U → G). However, if the (N)-Identifiers are group identifiers, the number of messages sent by the (N-1)-Entity will depend upon the number of (N-1)-Identifiers produced by the Group identifier mapping (either Group to Unique (G → U) or Group to Group (G → G)).

m : n - Many (N)-Identifiers per message to Many (N-1)-Identifiers per message. Both the (N)-Layer and the (N-1)-Layer allow many identifiers with each message. The number of messages sent by the transmitting

(N-1)-Entity depends upon the number of (N-1)-Identifiers produced by mapping the **m** (N)-Identifiers into (N-1)-Identifiers. The number of (N-1)-Identifiers produced (say **M**) depends upon the types of identifier used by the two layers. Therefore, the number of messages sent will depend upon **M** (the number of (N-1)-Identifiers produced) and **n** (the maximum number of (N-1)-Identifiers that can be associated with the (N-1)-Message):

M < **n** - only one message need be sent by the transmitting (N-1)-Entity since the number of (N-1)-Identifiers produced is less than the number of (N-1)-Identifiers that can be associated with the message;

M = **n** - only one message need be sent by the transmitting (N-1)-Entity since the number of (N-1)-Identifiers produced is equal to the number of (N-1)-Identifiers that can be associated with the message;

M > **n** - the ceiling of **M** / **n** messages will be sent by the transmitting (N-1)-Entity since the number of (N-1)-Identifiers produced are more than the number of (N-1)-Identifiers which can be associated with the message.

This layered view of multicast communications can be applied to many varied, yet important aspects of multicast communications such as intranetwork communications, internetwork communications, and message distribution within a Host.

3.1.2.1 Intranetwork Communications

In any layered architecture there will exist an (N)-Layer which has no supporting (N-1)-Layer. This (N)-Layer is the lowest layer of the architecture and is often referred to as the **Physical Layer** since it allows the physical connection of machines [ISO1981a]. A distributed system consisting of a single Physical Layer is said to support **intranetwork** communications (that is, communications on a single network). In an intranetwork communication it is assumed that all entities at a specific (N)-Layer support the same number and type of identifiers. Note however, that an (N-1)-Layer need not support the same number and type of identifiers as the adjacent (N)-Layer.

For example, a V-System Host consists of two layers: an application layer, consisting of Clients and Servers, and a kernel layer, through which all communication takes place (using an Ethernet as the underlying network). Both the application and kernel layers use One-Group multicast transmissions for the distribution of multicast messages. Since the members of a multicast set (consisting of Servers) are accessed with a One-Group transmission, the kernel layer is supplied with a single message and a single (Group) identifier, irrespective of the number of members of the multicast set. *The kernel transmits* the message and the supplied multicast set identifier to *all* other V-System Hosts on *the network using a single Ethernet broadcast* transmission. Clearly, from the discussion of layered multicast transmission, the One-Group to One-Group method of multicast transmission used by the V-System is highly efficient, requiring the minimum amount of message handling.

However, with a different underlying network, the V-System would not necessarily produce the minimum number of messages. Consider the following (hypothetical) example in which the Cambridge Ring is used in place of the Ethernet. In this example, the application layer would still use a single One-Group multicast transmission, however, the kernel layer would be forced to use One-Unique transmissions for the message to reach the intended Servers. This One-Group to One-Unique mapping would result in one or more transmissions of the message on the Cambridge Ring – clearly not as efficient as when using the Ethernet.

From these examples it is apparent that if the goal of a multicast implementation is to minimize the amount of message processing in a layered architecture, a method of multicast transmission should be chosen which minimizes the number of separate messages produced. This can be achieved in a number of ways, for example, by ensuring that the number and type of identifiers do not vary between the different layers.

3.1.2.2 Internetwork Communications

Distributed systems allowing communications between entities on different networks are said to support **internetwork** communications. The different networks in an internetwork communication are interconnected by **Gateways**. Unlike intranetwork communications, the transmitting and receiving entities on the different networks need not support the same number and type of identifier, even though they are at the same layer. Although this is not directly an inter-layer problem, the multicast transmission taxonomy can be used to describe the actions of a Gateway when interconnecting networks.

Consider, for example, the following situation: an (N)-Layer on one network supports One-Group multicast transmission, while a corresponding (N)-Layer on a separate network supports One-Unique multicast transmission. To permit a communication between these entities, the One-Group identifier must be converted into a list of one or more One-Unique identifiers. The identifier conversion can occur in a number of places, for example:

a) at the transmitting (N-1)-Layer. Individual messages must be sent to the receiving (N-1)-Entities through the Gateway, resulting in repeated message transmission from the transmitter's network, through the Gateway, to the Receiver's network;

b) at the intervening Gateway. By performing the identifier conversion at the Gateway, only one message need be sent from the transmitter to the Gateway. The Gateway would then be responsible for converting the One-Group identifier into a list of One-Unique identifiers.

As a further example, MP [Ahamad1985a] supports both intranetwork and internetwork multicast communication between Processes on UNIX 4.2BSD hosts. In MP, One-Group multicast transmission occurs at both the Process and UNIX-kernel layers – similar to that described in Section 3.1.2.1 regarding the V-System. However, when transmissions occur *between* networks, a single transmission is used to supply the Gateway with the message and a single Group identifier. The Gateway then generates the number and type of identifier expected by the receiving network using the supplied Group identifier.

If the goal of the multicast implementation is to minimize the amount of network traffic, the identifier type supported between the Transmitter and the Gateway should be chosen as to minimize the number of messages required. Network inefficiencies should *not* be allowed to spread across an entire internetwork – they should be kept local to a network.

3.1.2.3 Message Distribution

Another aspect of multicast transmission to be considered is the distribution of multicast messages by an (N-1)-Entity to the receiving (N)-Entities. Message distribution from the (N-1)-Layer to the (N)-Layer exhibits similar characteristics to that of multicast transmission from the (N)-Layer to the (N-1)-Layer and can be described using the multicast transmission taxonomy. For example, a receiving (N-1)-Entity may be expected to distribute messages to any receiving (N)-Entities using a single group identifier. However, if the receiving (N)-Entities can only be identified using individual unique identifiers, the receiving (N-1)-Entity may be forced to map the group identifier into a series of one or more unique identifiers before message distribution can take place.

This problem is common to all multicast implementations in which a lower layer is required to use a series of One-Unique transmissions to distribute messages to Entities in a higher layer. The problem has been recognized by several designers of distributed systems. For example, in [Cheriton1984b], a method of fast message distribution is described whereby 32-byte messages are distributed using some of the CPU's registers. Similarly, the Accent Distributed Operating System [Rashid1985a] uses a form of shared memory to reduce the amount of unnecessary message handling.

Clearly, the overall time required to deliver a message to all Destinations in a multicast set is affected by the time taken by the slowest Host to distribute its message.

3.1.3 Summary

This Section has shown the design, development, and possible usages of a multicast transmission taxonomy based upon:

a) the number and type of (N)-Identifiers used by a transmitting (N)-Entity in order to identify the receiving (N)-Entities, and

b) the mapping of the (N)-Identifier(s) (supplied by the transmitting (N)-Entity to the (N-1)-Server Layer) into (N-1)-Identifiers which can be used to identify the intended receiving (N-1)-Entities.

The usefulness of the multicast transmission taxonomy was demonstrated in several ways. First, the taxonomy was shown to facilitate the comparison of existing or proposed multicast transmission schemes. Second, the taxonomy proved useful in describing the amount of message handling required in layered architectures when, for example, performing intranetwork and internetwork multicast communications.

In addition, when using the taxonomy to describe layered architectures, the following observations were made:

a) multicast message handling within a layered architecture could be minimized if the different layers used the same type of identifier;

b) the number of messages generated in an internetwork multicast transmission is determined both by how the Receivers on the remote network are identified as well as whether the Transmitter or the Gateway is responsible for maintaining the list of Receiver identifiers;

c) the speed of a multicast transmission is determined by the number of receivers, their location, and the speed of message distribution, both over the network and within the Host.

These three points will be discussed further in Chapters Five and Six, when examining various multicast communication implementations.

3.2 A Multicast Response Taxonomy

A **multicast reception**, as defined in Chapter Two, consists of a Receiver receiving messages sent by many Transmitters. In this Section some of the problems associated with multicast reception using the Source-Destination model are considered.

In the Source-Destination model, the Source transmits a Request to set of multicast Destinations. The Source may or may not expect Responses from the Destinations. If Responses are *not* expected, the Source has no indication as to the success of the transmission (unless some subsequent communication takes place). However, if Responses *are* expected, the Source can, at best, only expect Responses from those Destinations which received the Request. Of the Responses that are received, different Sources will handle the Responses in different ways. For example, a Source may require:

- a single specific Response, with all other Responses being ignored;
- a certain number of Responses to be received;
- Responses from each Destination.

How Responses are handled is influenced by the type of Destination identification used by the Source. For example, if Destinations are uniquely identified, the Source has an exact indication as to the number and identity of the possible Destinations. However, if all members of the multicast set share a common (Group) identifier, the Source is forced to explicitly inquire as to the membership of the multicast set, since, it is assumed, a Group identifier does not normally offer any indication as to the number or identity of the Destinations.

3.2.1 Types of Multicast Response Handling

A basic division in any communication system in terms of Response handling is simply whether or not the Source expects a Response. It is possible to develop a simple table describing the actions of the Source based upon its Response expectations and the Response it actually receives (each member of the following table should be read as Source Expects - Source Receives, where **NR** indicates NoResponse and **R** indicates Response):

Source Actually Receives	Source Expects	
	No Response	Response
No Response	NR - NR	R - NR
Response	NR - R	R - R

Table 3-3: Unicast Response Handling

where the possible Response combinations are described as follows:

NR - NR: the Source does not expect a Destination to respond;

NR - R: the Source is not expecting a Response, but nevertheless receives one. This is an exception condition which can be resolved using a protocol;

R - NR: the Source, though expecting a Response to the Request, does not receive one. This again is an exception condition which can be resolved using a protocol;

R - R: the Source both expects and receives a Response to the Request.

In a multicast communication, Response handling is considerably more complex because there are more categories of possible Responses which need to be distinguished:

No Response: the Source expects no Destination to Response;

Single Response: the Source expects a Response from a single Destination;

Many Responses: the Source expects Responses from more than one Destination;

All Responses: the Source expects Responses from all possible Destinations.

The single communication table described above can be expanded to include the various types of Responses possible in a multicast communication. The Response handling types are based upon what the Source expects, while the actions within each type are dictated by what the Source actually receives:

Responses actually received by Source	Response Expected by Source			
	No	Single	Many	All
No	OK	None	None	None
Single	Unexpected	OK	Not-Many	Not-All
Many	Unexpected	Not-Single	OK	Not-All
All	Unexpected	Not-Single	Not-Many	OK

Table 3-4: Multicast Response Handling

where the contents of Table 3-4 should be read as follows:

OK: the Source received what it was expecting;

Unexpected: the Source is not expecting a Response, however at least one Destination returned a Response. This is an exception condition;

None: the Source is expecting some type of Response (one of Single, Many, or All) but no Response is received. This is an exception condition;

Not-Single: the Source is expecting a Response from a single Destination. If the number of Responses received is greater than one, this is an exception condition;

Not-Many: the Source is expecting Responses from Many Destinations. If the number of Responses received do not equal the number expected, this is an exception condition;

Not-All: the Source is expecting Responses from all possible Destinations. If the number of Responses received are less than the total number of Destinations, this is an exception condition.

The recognition and resolution of the exception condition could be handled by, for example, a protocol.

The above taxonomy is now examined in terms of two issues that can affect multicast Response handling: identifying Destinations and exception conditions.

3.2.1.1 Identifying Destinations

Although the multicast Response handling taxonomy described above enumerates all possible Response situations, it does not take into account the different types of Destination identification possible: Unique identification or Group identification.

If the Source uses Unique identifiers when transmitting a Request to the Destinations, it has available both the number of Destinations and their (Unique) identifiers. From this information it is clearly a trivial matter for the Source to determine which Destinations are responding - allowing all four possible Response types (No, Single, Many, or All) to be supported. For example, if Responses are expected from all Destinations but one does not respond, the Source can immediately identify the Destination in question since its identity is known.

However, the Source is at a disadvantage when using Group identifiers since it is not normally possible to determine the number or identity of the Destinations from a Group identifier. This means that the only types of Response that a Source using a Group identifier can be sure of are No-Response and Single-Response (assuming that at least one Response is received). It is not possible to guarantee the success of Many or All Response types since the total number of Destinations are not known.

Frank [Frank1985a] has proposed that by using a "static" list of Destinations (that is, a Group identifier is used in the transmission of the message, but the Source also maintains a list of the individual Destinations making up the multicast set), Many-Response and All-Response handling can be supported using group identifiers. Admittedly, this proposal does remove the possibility of unwanted or unexpected Destinations joining the multicast set, however it does not take into account the possibility of Destinations *leaving* the multicast set because of errors such as machine crashes.

Clearly, the designer of a multicast system is presented with a series of choices when considering the most efficient form of multicast reception. For example, additional protocols may be required when performing a One-Group transmission if All-Response handling is expected, since a list of unique identifiers would first have to be constructed before the communication could commence. However, in a One-Unique multicast transmission, the membership is already known, but the overhead of performing individual transmissions may be so great as to outweigh this potential advantage.

3.2.1.2 Exception Handling

The multicast Response handling taxonomy as presented at the start of this Section implies that the Source must handle all the Responses that are received. However, it is possible, in a layered architecture, to *filter* the incoming messages before they are received by the Source.

For example, a Source Process may require a specific Response from a set of Destinations - expecting all other Responses to be ignored. In order to offer such a feature to the Source Process, a lower layer would be required to filter the Responses, blocking all but the last.

In Section 3.2.2, exception handling and the filtering of messages will be discussed further with respect to specific networks, implementations, and applications.

3.2.1.3 Summary

In this Section, a taxonomy has been developed that can be used to describe the Response handling of a Source entity. The basic taxonomy is simply an enumeration of all possible combinations of Responses that the Source expects to receive: No Response, Single Response, Many Responses, or All Responses. In addition, the examination of this taxonomy has raised other issues with respect to multicast communications. For example, although a One-Group multicast transmission may require less transmissions than a One-Unique multicast transmission, additional processing may be required (in the One-Group transmission) since the list of members making up the multicast set may not be known and may have to be created before the communication can continue.

3.2.2 Multicast Response Handling Examples

The following Section uses the multicast Response handling taxonomy to describe multicast response handling in several different networks, implementations and applications.

3.2.2.1 Networks

From the description of the Ethernet given in Chapter Two, it should be apparent that an Ethernet Source is a No-Response type since an Ethernet packet is sent without provision for a response. (Note that if the Source uses an Ethernet Group address to communicate with the Destinations there can be problems when attempting to handle the Many-Response or All-Response types by a higher level of protocol, as suggested in Section 3.2.1.1).

The Cambridge Ring, as described in Chapter Two, allows communications involving a single Source and a single Destination. Since each transmission of a mini-packet is returned to the Source with a response indication (in the Response bits), it is clear that Cambridge Ring Response handling can be described as a Single-Response type. If a Response is not returned after a mini-packet is sent, the Source does nothing other than signal a higher layer of protocol that an error has occurred (each Cambridge Ring has a **monitor** node which is designed to catch any erroneous mini-packets and correct them [JNTIS1985a]). Any Responses that are generated by the Destination Process (as opposed to the Destination Cambridge Ring node) are returned in another series of mini-packets – these Responses are described by higher levels of protocol [Needham1982a].

A multicast communication on the Cambridge Ring can be obtained using any of several techniques. For example, it is possible to simulate a multicast transmission by a series of repeated One-Unique transmissions of the Request. This requires additional layers of software – where the lowest layer is simply performing a single Source to single Destination communication. The higher layers of software can implement different types of filtering as required by the application and described by the taxonomy.

3.2.2.2 Multicast Communication Implementations

Some of the (few) multicast implementations described in the literature are now examined. Reasons for this sparsity include the lack of understanding and development of protocols for multicast implementations [Gopal1984a] and the existence of Ethernet, which offers one particular form of multicast facilities – allowing designers to implement certain types of multicast transmission schemes quickly and easily.

In this Section, the different networks and distributed systems presented in Chapter Two are examined with respect to response handling.

3.2.2.2.1 The V-System [Cheriton1983a, Cheriton1984a]

The V-System is divided into two distinct layers: an application layer and a kernel layer. A Source application (or Client) can request one of three Response types from the kernel: No-Response, Single-Response, and Many-Response. Response handling requires the interaction of the Client with the kernel:

No-Response: A Request to which no Responses are expected is sent with a flag indicating to each Destination that Responses are not to be returned. This filtering is performed by the Destination kernel;

Single-Response: The first Response received by the Client's kernel is returned to the Client. All subsequent Responses are discarded by the kernel;

Many-Response: Each Response received is queued by the kernel. When the Client requires a Response, the kernel removes the first available Response from the queue (the onus is on the Client rather than the kernel to determine if the correct number of Responses have been received – a primitive exists which allows the membership of the multicast set to be determined). Should the Client transmit another Request to the same Group address, all queued Responses are discarded.

(Note: the All-Response type is not supported by the V-System kernel for two reasons. First, the overhead of an application supplying the kernel with a list of unique addresses or the kernel maintaining such as list was decided to be too complex. And second, the cost of retransmitting packets to non-responding Destinations was considered to be potentially too costly to implement directly in the kernel.)

The different types of V-System applications which have been described in the literature can be divided into two categories: Sources that do not expect Responses and Sources that do expect Responses.

The most widely publicized V-System No-Response type application is Amaze, a network game designed for multiple users on different machines [Cheriton1984a, Berglund1985a]. Briefly, the game consists of users moving "monsters" around their own screens, attempting to destroy other users' monsters. Each station transmits the status of its monster as a message to the other stations (sharing a group address). The transmitting station does *not* expect a Response to the message, since it is assumed that if a message is lost, another will be generated within a short period of time.

Other applications designed to test the Single and Many Response types have also been described. Single Response type applications have been developed allowing a Source to request a generic service and take the first Response supplied. More sophisticated applications have also been tested using the Many Response type handling. These include a Source transmitting a Request to a group of Destinations which return status information about themselves (such as processor load) as Responses. The Source can then choose a single Destination to communicate with based upon the information in the Response.

3.2.2.2.2 ASWE Serial Data Highway [Lakin1982a]

The ASWE Serial Highway supports a layered architecture which allows a Source Process to send a message to one or more Destination Processes. After a Request is sent, a Source Process can be expecting any of the four Response types. However, most applications only use the Single-Response type – with the Serial Highway interface filtering the returned Responses.

The services offered by the Destination Processes vary quite widely and include database managers, file servers, and processors performing specific mathematical functions. The most common way that a Source Process can access a particular service is to multicast to all the Destination Processes offering a particular service and accept only the first Response (taken as a "bid"). The Source then deals exclusively with the Destination Process that returned the first Response. Should the Destination prove unreliable or no longer provide the service, the Source Process can disregard the original Destination and reissue the Request.

3.2.2.2.3 Shoshin [Tokuda1983a]

Shoshin has a layered architecture with Source applications sending Requests to one, many, or all possible Destinations through the Source's communication manager. The communication manager applies filtering to the returned Responses based upon the type of Responses the application is willing to accept:

No-Response: since all transmissions suspend Process execution until a Response is received or an exception condition occurs, No-Response can be obtained by waiting for a Response from a non-existent Destination;

Single-Response: is possible by indicating to the communication manager which Destination the Response is expected from. All other Responses are discarded;

Many-Response: can be implemented in two separate ways. First, it can be implemented in a simple first-come, first-served fashion, where each Response is queued by the communication manager until required by the application. And second, it is possible to exclude a specific Destination's Response and accept Responses from all other Destinations.

The All-Response type is not available on the Shoshin system. (Although not discussed explicitly in the Shoshin paper, it is assumed that a multicast communication uses a Group address, making the handling of the All-Response type extremely difficult.)

3.2.2.2.4 MIKE [Tsay1983a]

Although the main emphasis of the research on MIKE is in the development of guardians for distributed processors, the underlying network does permit multidestination communications:

No-Response: a Source can send a message to multiple Destinations without expecting a Response;

Single and Many Response: are referred to as "reliable" communications. That is, a message is sent to several Destinations and Responses are expected from all those that are currently active. Destinations that do not return Responses within a timeout period are assumed to be inactive;

All-Response: are referred to as "guaranteed" communications. In this situation, MIKE ensures that every Destination that is to receive a copy of the Request receives one and returns a Response – there is no time limit imposed upon the Destinations.

In order to allow all four possible Response types to be supported, MIKE must either use a combination of group and unique addresses or lists of unique addresses to represent Services. Unfortunately, this is not made apparent in the MIKE paper.

The authors do not describe any multicast applications which they have specifically examined during their testing.

3.2.2.2.5 Other Implementations

In [Gopal1984a], the authors do not describe an implementation, but rather the design of a system which allows a Request sent to several Destinations to be acknowledged as efficiently as possible. The basic algorithm assumes that the Source sends a series of Requests, each with a unique sequence number. The Destinations return acknowledgments (as Responses) if the Requests were accepted, otherwise negative acknowledgments are returned (Responses must be returned within a timeout period, otherwise any missing Responses are assumed to be negative acknowledgments). When a negative acknowledgment (or a timeout) is received, the Source performs a "Go-Back-N" retransmission algorithm, which means that all Requests, starting with the one that was not received correctly, are retransmitted.

Three different types of "Go-Back-N" (GBN) algorithms are proposed. The first, Memoryless GBN, requires that all Destinations respond with an acknowledgment to each of the retransmitted Requests - requiring that the Source handle the All-Response type. The second, Limited Memory GBN, expects acknowledgments only from those Destinations that did not acknowledge the retransmitted message (the Source is to handle either Single or Many Response types). However, all other retransmitted Requests must be acknowledged by all the Destinations - once again making the Source handle the All-Response type. Finally, there is Full Memory GBN, which accepts Responses only from those Destinations that have not yet responded to any outstanding Requests. In this case, the Source is to handle Single, Many, or

All Response types, depending upon the number of acknowledgments still outstanding for each Request.

There have been other multicast communication implementations described in the literature, however, the problems and applications of multicast Response handling have, for the most part, not been addressed by their authors. For example, three recent papers on multicast communications either mention Response handling only in passing [Ahamad1985a] or not at all [Chang1984a, Frank1985a].

In each of the above three papers, the underlying network was assumed to be an Ethernet, and in all cases, Group identification was used. Therefore, at the lowest layer, the Ethernet would be offering a No-Response type of Response handling - any other type would have to be incorporated into a higher level of protocol. Note however, the problems that were described in Section 3.2.1.1. regarding the identification of Destinations using Group identifiers would exist in each of these implementations.

The UNIVERSE Project [Leslie1984a, Waters1984a], which attempts to link several Cambridge Rings by a satellite broadcast channel, is another case of the designers describing multicast transmission but not Response handling. However, from the description of the Cambridge Ring (see Section 3.2.2.1 and Chapter Two), it is clear that at the Source's interface to the Ring it must be a Single Response type. Once again, at higher layers, a protocol could present a different Response handling interface to the application.

3.2.2.3 Applications

In Chapter Two, four applications were presented as "typical" examples of multicast communication requirements. In this Section, each of these application are now discussed in terms of the multicast response taxonomy.

The first application, the "time-signal generator", consisted of the Time-Signal generator transmitting time-signals to any Process which required a time value. Since no responses were expected by the Source, the Time-Signal generator was clearly a No-Response type application.

The second application involved a Source Process transmitting a request to a multicast set consisting of one or more Time-Servers. The Source then waited for responses, the first of which was accepted. To the original Source Process, since only one message was returned, this was clearly a One-Response type application. Note however, that to the lower layers responsible for the filtering of the incoming Responses, this could be a Many-Response or All-Response type application.

The third application, the TMR File-Server, is an example of a Many-Response type application since the application will continue as long as there are two or three (i.e. many) Destinations responding.

The Two-Phase Commit Protocol, the final application example, required all the Destinations to respond with indicators as to the success (or failure) of the invocation of the different phases. Since the Source expects each Destination to respond, the Two-Phase Commit Protocol is an example of an All-Response type application.

3.2.2.4 Summary

This Section has presented a taxonomy which can be used to describe the actions of a Source after it has transmitted a message to one or more Destinations. It was shown that *how* Destinations are identified (using either Unique or Group identifiers) has a major impact on the type of Response handling available to the Source. Destinations accessed by Unique identifiers permit the Source to handle any of the four different Response types. However, since, it is assumed, the number of Destinations and their identity cannot easily be established from a Group identifier, only the No-Response and Single-Response types (and in some cases, Many-Response) could be used with Group identifiers without the development of additional protocols.

The multicast response taxonomy also demonstrated the need for different types of message filtering by the lower layers of a Receiver. For example, it is possible for one layer to perform one type of Response handling and a lower layer to perform another. This was shown in all of the Response handling implementations examined, the lowest layer handled Many or All Response types, while at a higher layer, an application may be expecting, for example, the No-Response or Single-Response type.

The taxonomy also proved useful in describing the response handling capabilities of networks, distributed systems and various applications. From this type of information it should be possible for the designer of an application requiring multicast communications to determine the type of filtering required.

3.3 Multicast Communications in Layered Distributed Systems

In Chapter Two, a means of describing Process-to-Process communications using a simple layered architecture model was presented. In this model, the transmitting Process transmits its message with the Host-Port identifier associated with the receiving Process. By enumerating different Host-Port identifier combinations, it is possible to develop a taxonomy for describing the effect of using Host-Port identifiers for multicast transmission. In addition, it is possible to use the Host-Port enumeration to describe the probable amount of message handling required by a receiving Communication Layer.

In this Section, these taxonomies are developed and discussed in terms of several existing layered architecture implementations.

3.3.1 Host Identifiers

By using the multicast transmission taxonomy developed in Section 3.1 and applying it to a Host in a layered architecture, one finds the following:

a) the Transmitter is a single Host;

b) the Receiver is one (or more) Host(s);

c) the communication services are supplied by a Network;

d) Receivers (Hosts) are identified by some type of **Host identifier**, often referred to as a **network address**.

The efficiency of a Network Layer multicast transmission is dictated (in part) by how the receiving Host(s) are identified. To allow a Host to be identified on a Network, all Networks support at least one of the following three types of Host identifier:

Unicast: identifying a single Host. A message transmitted with a Unicast Host identifier is received by at most one Host (i.e. it is a One-Unique transmission);

Multicast: identifying a set of Hosts, sharing a common identifier. A message sent with a Multicast Host identifier is a One-Group transmission in which only those Hosts belonging to the Group receive a copy of the message;

Broadcast: identifying all possible Hosts on a Network. A message sent with a Broadcast Host identifier is another One-Group transmission where the Group encompasses *all* Hosts.

In terms of transmission efficiency, using a Multicast or Broadcast identifier may be more efficient than using a Unicast identifier, since a message sent with a Multicast or Broadcast identifier (ideally) need only be sent once, whereas a message sent with a Unicast address must be sent separately to each receiving Host (as shown by the multicast transmission taxonomy).

3.3.2 Port Identifiers

It is also possible to use the multicast transmission taxonomy to describe the identification of Process(es) using Ports:

a) the Transmitter is a Process on a Host;

b) the Receiver(s) are one (or more) Process(es) residing on one (or more) Host(s);

c) the communication services consist of the underlying Communication and Network Layers on the transmitting and receiving Hosts;

d) Processes are identified, indirectly, by some type of **Port identifier**.

As in the case of Host identification, the efficiency of the communication is dictated by how the receiving Ports are identified. Four methods of identifying a Port are proposed:

Unique: the Port identifier is a Unique identifier, unique to *one* Process on *one* Host. The identifier does not exist on any other Host;

Shared: the Port identifier is a Unique identifier, accessible by at most *one* Process on a Host. However, the identifier can exist on *all* Hosts on the network;

Local: the Port identifier is a Group identifier, which can be accessed by *any number* of Processes on *one* Host. The identifier does not exist on any other Host;

Global: the Port identifier is a Group identifier, which can be accessed by *any number* of Processes on *all* Hosts.

There are advantages and disadvantages to each of the Port identification methods described here. Unique Port identifiers can be costly in terms of the number of messages that the Communication and Network Layers, on both the transmitting and receiving Hosts, are expected to handle. For example, if

Unique Port identifiers are used and a Host supports "n" Processes, all of which belong to a particular multicast set, then, in the worst case, the receiving Communication Layer can be expected to handle n copies of the message.

Although using Shared, Local, or Global Port identifiers can decrease the number of transmissions required, there is the disadvantage that the transmitting Process has no indication as to the number of receiving Processes. This can produce a variety of problems should the transmitting Process ever expect messages to be returned by the receiving Process(es), as for example, in the Source-Destination model described in Section 3.2.

3.3.3 Host-Port Identifiers and Multicast Transmission

In this Section, the number of message transmissions required by a transmitting Communication Layer are discussed in terms of Host-Port identifiers. First, the maximum number of transmissions required for a message to reach all members of a multicast set are considered for all Host-Port identifier combinations (assuming no retransmissions). This is followed by an examination of how the different Port identifiers can be used to support the minimum number of transmissions to the members of a multicast set. In each of the following cases, it is assumed that there are N possible receiving Processes.

Clearly, the maximum number of message transmissions required by the transmitting Communication Layer occurs when each receiving Process in the multicast set is associated with a Unique Port, irrespective of how the Host is identified. In this situation, the transmitting Communication Layer must send "N" copies of the message, one to each receiving Process. However, in a network which allows *many* identifiers to be associated with a message, the number of transmissions can decrease to (ideally) one, as shown by the basic multicast transmission taxonomy in Section 3.1.1.

Ideally, the transmitting Communication Layer should only transmit one message which is then received by the various Processes making up the multicast set. However, the number of transmissions can vary from one to "N", depending upon the how the Host-Port identifier identifies the intended receiving Processes.

3.3.3.1 Unique Ports

To achieve the minimum number of message transmissions when dealing with a multicast set consisting of Processes which can only be accessed using a Unique Port identifier, the multicast set could only have one member, irrespective of how Hosts are identified since at most one Process can be associated with a Unique Port identifier. However, if the network permits *many* Host-Port identifiers to be associated with each message, the number of messages to be transmitted could be decreased. Further reductions could be achieved by using a Multicast or Broadcast Host identifier with multiple Port identifiers, since the transmitting Communication Layer would not be expected to transmit to each Host individually.

The problems associated with receiving messages on Hosts which do not support members of the multicast set after a transmission with Multicast or Broadcast Host identifiers are discussed further in Section 3.3.4.

3.3.3.2 Shared Ports

A Shared Port can exist on any number of Hosts, but like the Unique Port, only one Process on each Host is allowed to access it. Therefore, in order to minimize the number of message transmissions on the part of the transmitting Communication Layer, all receiving Processes should reside on separate Hosts and be accessed by a Multicast or Broadcast Host identifier.

However, if more than *one* Process resides on the *same* Host, the transmitting Communication Layer is forced to send multiple messages (unless the network allows many identifiers per messages, in which case the number of transmissions depends upon the number of identifiers that can be associated with the message). In both of these situations, the number of Process identifiers depends upon the maximum number of receiving Processes on a Host and whether their Port identifiers are Shared or Unique. For example, on a network allowing a maximum of one Port identifier to be sent with each message, if four Processes belonging to the multicast set reside on two Hosts (two Processes per Host), the minimum number of transmissions would be two - occurring when each Process was using a Shared Port (i.e. sharing it with one other Process on the other Host), whereas the maximum number of transmissions would be three: one, using a Shared Port identifier and the other two, using a pair of Unique Port identifiers.

3.3.3.3 Local Ports

A Local Port identifier allows *many* Processes to share a Port on a *single* Host; the Port identifier does not exist on any other Host. Clearly, in this situation, the transmitting Communication Layer need not send more than one message if all the receiving Processes reside on a single Host. It would be sufficient to use a Unicast Host identifier in this situation - thereby avoiding unnecessary message reception by Hosts not supporting Local Port identifiers.

However, should the receiving Processes reside on different Host, the transmitting Communication Layer is forced to either transmit a series of messages with Unicast Host identifiers, one to each Host which supports members of the multicast set or to transmit message with a Multicast or

Broadcast Host identifiers if the network allows many identifiers to be associated with each message.

Unlike Shared Ports where the number of transmissions is dictated by the number of Processes on a single Host, the number of transmissions using a Local Port identifier is determined by the number of receiving Hosts which support members of the multicast set.

3.3.3.4 Global Ports

A Global Port identifier can be associated with any number of Process on any number of Hosts. The number of transmissions by the Communication Layer depends, in part, upon the type of Host identifier supported. For example, if the network only allows Unicast Host identifiers, Global Ports are degraded into Local Ports.

Ideally, Global Ports should be used with Multicast or Broadcast Host identifiers, since only one transmission would be required. However, as in the other Port types, if members of the multicast set happen to use different Global Port identifiers, the transmitting Communication Layer may be required to make additional transmissions.

3.3.4 Host-Port Identifiers and Multicast Message Handling

From the discussion on Host-Port identifiers and multicast transmission in the previous Section, three broad categories for describing Host-Port identifiers and how they affect message distribution by a receiving Communication Layer are proposed:

Repeated - any receiving Communication Layer which must receive more than one message in order to ensure that all possible Process(es) receive a copy of the message can be classified as **Repeated**. For example, any Communication Layer handling messages supplied with a Unique Port identifier (that is, Unicast-Unique, Multicast-Unique, and Broadcast-Unique) may be classified in this category if there is more than one receiving Process on the Host in question;

Unnecessary - a receiving Communication Layer which receives a message for a Port that it does not support can be classified as **Unnecessary**. Any message supplied with a Broadcast Host identifier may be classified in this category. This category exists because of the nature of the Broadcast identifier – all Hosts, irrespective of the Processes they support, receive a copy of a message sent with a Broadcast Host identifier.

Note, this situation can also arise if a Port which is normally available on a Host is unavailable for some reason. Should this occur, the receiving Communication Layer can be classified as "unnecessary" since there is no Port to direct it to;

Required - this category occurs when just one message is received by the receiving Communication Layer and is required by one or more Processes. Clearly, in this category, if there is only one possible receiving Process, all Host-Port identification combinations can be classified in this category.

However, should the number of receiving Processes be greater than one, then only Local and Global Port identification ensure that the receiving Communication Layer receives no more than one copy of the message (assuming that all Processes access the same Port).

3.3.5 Host-Port Identification Examples

In this Section, a series of examples using the two Host-Port classification models developed in Section 3.3.3 and 3.3.4 are presented, describing how multicast communications between Processes on different Hosts could be realized using existing layered architectures.

3.3.5.1 Cambridge Ring Ports

The Cambridge Ring **Packet Protocol** [Banerjee1985a, Panzieri1985a] allows a transmitting Process to identify a Port on the receiving Host (note, this assumes that a Host is equivalent to a station). The Port is associated with a Process. A Port number is not unique to a Host, that is, many different Hosts on the network can use the same Port Number. However, the Processes associated with these Ports on the different Hosts need not receive the same multicast transmission, because of the unicast nature of the Cambridge Ring.

For a communication to take place on the Cambridge Ring using the Cambridge Ring packet protocol, the Source Host can only supply a unicast station (Host) address and a Port number. Port numbers can be Unique to a Host or (in theory) be Shared across all Hosts on the network receiving the same multicast transmission.

For a multicast communication therefore, when using the Cambridge Ring Packet Protocol, a total of N transmissions of the same packet will be required (that is, one for each Process - or simply a series of Unicast-Unique transmissions). Alternately, each receiving Process could be made to reside on a separate Host; if each potential receiving Process accessed the same Port number, the packet protocol could be described as Unicast-Shared. Note that the same number of transmissions are required on the part of the transmitting Host, irrespective of whether the Port is Unique or Shared.

3.3.5.2 Ethernet Sockets

The Ethernet is designed for Unicast, Multicast, and Broadcast Host identification. Several successful Process identification schemes have been built on top of the Ethernet, such as PUP [Boggs1983a] and UNIX sockets [Leffler1983a]. A socket Port cannot be shared on a Host, meaning that Process-to-Process transmissions must use Unique Port identifiers. However, since a Port can be "well-known" and shared by Processes on different Hosts; and the Ethernet supports Broadcast Host identification, sockets can be described as offering Broadcast-Shared identification, since only one Process per Host may receive a copy of the message.

Ahamad and Bernstein [Ahamad1985a] have modified the existing UNIX socket software to allow multiple Processes on the same Host to receive copies of a multicast message (a Broadcast-Local implementation). However, if the Port is made available on all machines, this version of sockets can be described as Broadcast-Global.

Finally, the problems associated with unnecessary reception of messages can be eliminated in all but a few cases by using specialized Ethernet hardware which supports Multicast-Host identification. This hardware [DEUNA1983a] permits an Ethernet station to selectively receive up to ten different (Multicast) Host identifiers. Although no implementations have been described in the literature, the DEUNA hardware potentially permits the design of Multicast-Global identification.

3.3.5 Host-Port Identifiers and Multicast Sets

A common requirement of many of the applications discussed in Chapter Two was the ability to *send* a message to one or more Receivers. In all cases, the Transmitter not only sent the message, but *identified* (implicitly, if not explicitly) the intended Receivers. In this Section, one of the problems associated with multicast sets in terms of Host-Port identifiers, notably how multicast sets are formed, is examined.

There are two ways in which a Process can become a member of a multicast set. First, the Process can *ask* to join an existing multicast set (several examples of applications in which Processes join multicast sets have already been discussed). How this facility is offered is an implementation problem – for example, a multicast set might have a predefined identifier, which the Process could simply assume, or a Process could join a multicast set by informing all possible Source Processes (or a series of name servers) of its existence.

A Process can also *be asked* to join a multicast set. In this situation it is assumed that there exists another Process which requires the creation of a new multicast set or wishes to increase the membership of an existing set.

As shown in this Section, for a multicast transmission to occur, the Source Process must have access to one or more Destination Host-Port identifiers. For an optimal multicast transmission, the Destination Process(es) should be identified using either a Multicast-Global or Broadcast-Global identifier. Therefore, it is *not* sufficient for Processes about to join a new multicast set to return their own unique Host-Port identifiers – as this would probably increase the number of transmissions required since the various Destination Processes would be identified using Unique Port identifiers.

Ideally, the Process creating the new multicast set should generate a new, unique Global Port identifier which could be supplied to the new members of the multicast set and be used to identify them in subsequent transmissions. The Global identifier must be unique in order to avoid having messages arrive at the wrong Destination Processes. The generation of a new Global Port identifier to uniquely identify the new multicast set may be easier said than done – for example, it implies that the Port number is not already in use. (The problems associated with generating unique identifiers for use as multicast set identifiers will be returned to in Chapters Four and Five.)

However, the Source Process often has less control over Host identifiers than it does over the Port identifier since, in some distributed systems, Hosts can only be accessed using *unique* Host identifiers. At the other extreme, certain Hosts may support hardware which allows them to recognize a number of Host identifiers, implying that, for example, the Host identifier could be made identical to the Global Port identifier, allowing Multicast-Global Destination identification.

Whatever method of Host identification is used, it must be conveyed back to the Source Process to permit the creation of a list of the Host-Port identifiers making up the new multicast set. This list can be used by the Communication Layer when messages are to be sent to the members of multicast set.

The following example demonstrates some of the requirements for the creation of a new multicast set by a Source Process and a series of Destination Processes. The Source Process algorithm is as follows:

```
GetANewGlobalIdentifier;
SendRequestToPotentialNewMembers;
WaitForResponses;
CreateNewMulticastSet;
SendConfirmationMessageToNewMembers;
```

The Source Process (the creator of the new set) must send a message to those Processes which may be able to become members of the new multicast group. (To allow this, it is assumed that a series of Processes exist which can join new multicast sets when requested.) The Source Process, after transmitting a message with the associated Global Port identifier requesting Processes to join a new multicast set, waits for responses from the members of the multicast set which can join the new multicast set. Once the responses are available, the Source Process can create a list of the (Destination) Host-Port identifiers. The size of this list depends upon how Destination Processes are identified. Ideally, the list contains one entry, a Multicast-Global (or Broadcast-Global) identifier. However, in the worst case, it contains a series of Unicast-Unique identifiers, one for each Destination Process. Once the multicast set has been created the Source Process may then transmit messages to members of the new set – the first message of which should be a confirmation of the new member's membership in the new multicast set.

The following describes a Process which can join a new multicast set:

```
JoinTheCanBecomeAMemberSet;
repeat
    WaitForARequestToJoinANewSet;
    SendBestHostAndProcessIdentifier;
    JoinTheNewMulticastSet;
    WaitForConfirmationOfMembership;
    if Confirmation < > Yes then
        LeaveNewMulticastSet;
until
    Confirmation = Yes;
WaitForSubsequentMessages;
```

The Destination Process (i.e. a Process which can join a new multicast set) must first join a multicast set which will receive requests to become a member of a new multicast set. Once joined, the Process waits indefinitely until a RequestToJoinANewSet message (which includes the new Global Port identifier) is made. The Destination Process then responds to the Source Process with an indication that it is willing to join and its *optimal* Host-Port identifier (ideally this should be a Broadcast-Global or Multicast-Global identifier). To allow the Destination Process to determine whether or not it has been accepted into the new multicast set, it joins the new set and waits for a Confirmation message. The Confirmation message is expected to arrive within a certain time period, otherwise the Destination Process assumes that it was not included in the set and leaves. If the Process is accepted into the set, it stays as a member of the new multicast set and waits for subsequent messages.

3.3.6 Summary

In this Section, it was shown that a multicast communication in a distributed system is affected by the type of:

a) Host identification used by a network when supporting Host to Host communications, and

b) Port identification used by the Communication Layer when supporting Process to Process communications.

By combining the different types of identification (i.e. Host and Port), it was shown which pairs of combinations are the most efficient in terms of multicast transmission. From the most to least efficient, these are:

a) Multicast-Global or Broadcast-Global: in which the transmitter need only send one message which is received by the intended Hosts. The receiving Communication Layer can keep the amount of message handling to a minimum since the intended Process(es) all share the same Port identifier;

b) Unicast-Global, Unicast-Local, Multicast-Local, or Broadcast-Local: in which the transmitting Host is required to send many messages, but only one to each receiving Host. Only one message need be sent to each Host, since at each Host, should there be multiple Processes, these can all be accessed using the Local (or Global, in the case of Unicast-Global) Port identifier. The number of transmissions depends upon the number of Hosts;

c) Multicast-Shared or Broadcast-Shared: if all Destination Processes are on separate Hosts, the transmitting Communication Layer need only send one message. However, in the worst case, when all Destinations reside on the same Host, the transmitting Communication Layer is forced to send individual message to each Process;

d) Unicast-Unique, Multicast-Unique, Broadcast-Unique, or Unicast-Shared: the worst case is the transmission of a message to receiving Process(es) which must be accessed using a Unique Port identifier. In these situations, each message must be sent repeatedly with a different Port identifier, even if the receiving Process(es) happen to reside on the same Host. The number of transmissions depends upon the number of Processes and the number of identifiers that can be associated with a message.

In addition, the amount of message handling required by a receiving Communication Layer can be discussed using the Host-Port paradigm since message handling is also affected by the type of Port identification, and in some cases, the type of Host identification used. For example, a receiving Communication Layer will perform the minimum amount of message handling when it need only receive one message to ensure that all the intended Processes receive a copy.

Finally, by using the Host-Port paradigm, it was possible to describe some of the requirements of multicast set management. Specifically, the ability to:

- **generate** a new Global Process identifier, which can be used to identify the members of the new multicast set;
- **create** a new multicast set, based upon the Host and Process identifiers supplied by the members of the new set;
- to supply the **best** Host-Port identifier to the Process creating the new multicast set to ensure that the most efficient means of multicast communication are used.

From this discussion of multicast identifiers, one can conclude that two ways of reducing the cost of a multicast communication are:

a) maximize the number of receivers represented by a single identifier;

b) minimize the number of messages supplied to the Communication Layer by the Transmitter.

3.4 Concluding Remarks

In this Chapter, by using some of the basic divisions of communication, notably transmission and reception, a set of multicast communication taxonomies were developed.

Two multicast transmission taxonomies were presented. The first, based upon the Transmitter-Receiver model, was used to describe the different types of identifier handling required in a multicast transmission. The second allowed the enumeration of different types of identifiers for multicast communications in (layered) distributed systems using Host-Port identifiers.

A multicast reception taxonomy was also presented. The reception taxonomy outlined the different types of message handling possible and its effects on a receiving Process.

In all of the taxonomies presented, it was shown that the type of identifier used could greatly affect the performance of the communication. In subsequent Chapters, these classification schemes and the different types of identifier will be used to develop and implement a set of multicast communication primitives.

Chapter 4 Multicast Primitives

For a distributed system to support multicast communications, facilities should exist at the various layers (such as the Process Layer or the Network Layer) which permit multicast communications. For example, if the applications presented in Chapter Two were to be implemented on a distributed system supporting multicast communications, one would expect to find a set of common facilities which would permit a standard form of multicast communication between, say, Processes.

In this Chapter, a series of nine such facilities or **primitives** are presented and it is shown how they can support Process-to-Process multicast communications. These primitives are intended, for the most part, to be unrelated to any specific network or distributed system. Chapters Five and Six will include examples of how these primitives could be implemented on a variety of networks and distributed systems.

This Chapter is organized as follows. In Section Two, the proposed multicast communication primitives are discussed in terms of interprocess communications. Several examples of how the primitives can be used are presented in Section Three. In Section Four the proposed primitives are compared with other existing multicast communication primitive implementations. The Chapter is concluded with a review of the proposed primitives.

4.1 Multicast Primitives

In Chapters Two and Three, various requirements were presented which described some of the features that should be supported in a distributed system if it was to permit multicast communications. For example, in Chapter Two it was shown that facilities should exist to allow a Process to send a single message to the members of a multicast set, while in Chapter Three the need to support different types of filtering was demonstrated. In this Section, these and other features which motivated the choice of a set of general purpose interprocess multicast communication primitives are expanded upon.

The examination of the primitives is divided into two broad categories:

a) multicast set management primitives, and
b) multicast communication primitives.

4.1.1 Multicast Set Management Primitives

In this Section, a set of primitives for supporting the management of multicast sets are proposed. Specifically, this Section examines how a Process can join or leave existing multicast sets as well as how new multicast sets can be formed. However, prior to examining the primitives, the different types of identifier to be supported are discussed.

4.1.1.1 Identifiers

A requirement of many of the applications discussed in Chapter Two was the ability of a Transmitter to transmit a message to one or more Receivers. In all cases, the Transmitter not only sent the message, but also identified the intended Receivers either explicitly or implicitly. In addition, from the examination of identifiers in Chapter Three, it is proposed that the multicast communication primitives should support the following types of identifier:

a) an identifier which represents a **individual** Transmitter or Receiver. For example, in a distributed system, a Process could be identified using a Unique identifier such as a Unicast-Unique Host-Port identifier. This identifier type allows a Process to transmit a message to a specific Process or a receiving Process to identify the Process transmitting a message. This type of identifier will be referred to as a **unicast identifier**;

b) an identifier which identifies all the members of a multicast set. This single (Group) identifier or **multicast set identifier** is equivalent to an alias or shorthand method of identifying the members of the set. The multicast set identifier will be used in two ways. First, a transmitting Process can use it to send a message to a set of Processes belonging to a multicast set, and second, a Process can use it to indicate the multicast set to which it belongs.

The exact format of the proposed identifiers is an implementation detail which will be returned to in Chapter Five. However, for the purposes of this Chapter, it will be assumed that both unicast identifiers and multicast set identifiers are numeric.

4.1.1.2 Accessing Existing Multicast Sets

Clearly, before a multicast communication can take place, both the Transmitter *and* the Receiver must obtain a common multicast set identifier. In this Section, three primitives are presented which are necessary to allow access to existing multicast sets (an "existing" multicast set is assumed to be a multicast set which already has a multicast set identifier associated with it).

4.1.1.2.1 Obtaining an Existing Multicast Set Identifier

In many distributed systems, application services are often referred to by textual **names**, primarily for the benefit of (human) users of the system [Shoch1978a]. Although the names used may vary, they often refer to the same application or service (for example, a time-server may be called "Clock" by one user but "TimeServer" by another). To allow users to identify applications by a textual name rather than by multicast set identifier, the *getid* primitive is proposed:

ReturnCode := *getid* (Name, var Identifier)

where:

Name: is the character string which identifies the multicast set;

Identifier: is the identifier associated with the supplied name and is returned by the *getid* primitive. The identifier can be either a unicast identifier or a multicast set identifier;

ReturnCode: an indication as to the success or failure of the *getid* primitive.

A "null" Name (i.e. a string of zero length) causes *getid* to return the unicast identifier of the Process which invoked the primitive. This is intended to allow a Process to receive messages destined to itself.

Should the supplied name not exist, *getid* is to return an error code of "NameNotFound" otherwise it returns a code of "Success".

Note that *getid* is not restricted to multicast set identifiers. For example, the Name could be associated with a unicast identifier, identifying a single Process. The association of Name and Identifier is discussed further in Section 4.1.1.2.3.

For the remainder of this Chapter, it is assumed that a **name server** exists which maintains the list of Names and related Identifiers.

4.1.1.2.2 Joining a Multicast Set

Ideally, once a Process has obtained a multicast set identifier, using, for example, the *getid* primitive, it should be able to either transmit messages to the multicast set or receive messages destined to a particular multicast set.

However, there are situations where a Process must announce that it wishes to become a member of a multicast set before it can receive messages intended for the multicast set in question.

For example, if the Process about to become a member of a multicast set can only be identified with a unicast identifier, all potential transmitting Processes must be informed of the identifier associated with the new member of the multicast set. The same problem can exist on a network which supports an efficient form of multicast transmission (such as One-Group), but uses a distribution facility to supply messages to the members of the multicast set within the Host. In both of these situations, the receiving Process should have a mechanism whereby it can inform the facilities supporting multicast communications that it expects to receive messages destined to a particular multicast set.

To overcome this problem, a *join* primitive is proposed which allows a Process to join a specific multicast set. How the *join* primitive is implemented is clearly dependent upon the lower layers supporting multicast communications; however, at the Process Layer, the primitive is simply:

ReturnCode := *join* (Identifier)

where:

Identifier: an identifier (assumed to be a multicast set identifier), indicating which multicast set the Process wishes to join;

ReturnCode: a value indicating the success or failure of the *join* primitive (see below).

If the supplied Identifier is not a multicast set identifier or, for some reason, the multicast set could not be joined, the *join* primitive returns a ReturnCode of "CannotJoin", otherwise "Success" is returned.

There is no limit to the number of multicast sets that can be joined by a Process. Note however, messages are only made available when the *receive* primitive is invoked (see Section 4.1.2.2).

4.1.1.2.3 Leaving a Multicast Set

Once a Process has performed whatever tasks are required of it and there is no need for it to remain a member of a multicast set, it is assumed that the Process can leave the multicast set. As in the case of the *join* primitive, a *leave* primitive is proposed to assist in the maintenance of the membership of the multicast set:

ReturnCode := *leave* (Identifier)

where:

Identifier: an identifier (assumed to be a multicast set identifier) indicating the multicast set which is to be left;

ReturnCode: a value indicating the success or failure of the *leave* primitive (see below).

When a Process has left a multicast set, subsequent messages will not be made available (i.e. the *receive* primitive will fail if called with this multicast set identifier).

A ReturnCode of "NothingToLeave" is returned if the identifier is not a multicast set identifier or the multicast set has not been previously joined, otherwise a ReturnCode of "Success" is returned.

4.1.1.3 Creating New Multicast Sets

The primitives described in the previous Sections all deal with *existing* multicast sets. Clearly, there are situations where new multicast sets are required – for example, the addition of a new service, unrelated to any already existing in the distributed system. In these situations, not only are new members required, but also new, unique multicast set identifiers.

In this Section, a discussion of several aspects of multicast set membership, notably the creation of new multicast set identifiers, the identification of multicast set members and the association of names with multicast set identifiers is presented.

4.1.1.3.1 Generating New Multicast Set Identifiers

As it stands, no primitive has yet been proposed which allows the generation of new multicast set identifiers – the *getid* primitive simply returns an existing unicast identifier (identifying a single Process) or multicast set identifier (identifying an existing multicast set).

However, when a new multicast set is formed, it requires a unique multicast set identifier if confusion over multicast set membership is to be avoided. A single primitive is available for this operation:

newid (var Identifier)

where:

Identifier: is a new unique identifier returned by the *newid* primitive which can be used as a multicast set identifier.

The identifier returned by the *newid* primitive must be unique within the context of the network in which it is used. Various methods exist for the creation of unique identifiers, for example, the concatenation of a Process's unicast identifier with the current time-of-day.

4.1.1.3.2 The Identification of Multicast Set Members

Thus far it has been assumed that the various members of a multicast set are identified with a single multicast set identifier. This assumption has been made to ensure that in a multicast transmission a Process sends no more than one message. However, as shown in the discussion of Host-Port identifiers in Chapter Three, not all distributed systems support the concept of a single identifier identifying a set of receiving Processes. For example, in many distributed systems, the single multicast set identifier supplied by the transmitting Process many have to be expanded into several Host-Port identifiers by the underlying Communication Layer. This mapping (from multicast set identifier to a series of Host-Port identifiers) should be as efficient as possible, generating the minimum number of Host-Port identifiers in an attempt at reducing the number of messages being transmitted.

By way of an example, consider a distributed system which supports both Unicast-Host and Broadcast-Host identification but allows at most one Process to be associated with a Port. A Process attempting to set up a new multicast set may request the potential members to associate themselves with the Host-Port pair <Broadcast, 2000> - thereafter all messages would be broadcast to Port 2000. However, should Port 2000 already be in use or two Processes on the same Host attempt to join the multicast set, but only one actually succeeds in accessing Port 2000, the Process unable to access the specific Port should have a mechanism to inform the Source that it could not be associated with Port 2000, but could access (say) Port 2001. The Source Process, upon receiving the responses could then create a membership list consisting of the identifiers <Broadcast, 2000> and <Unicast, 2001>, causing the *send* primitive to send two messages with each transmission, one to the broadcast address (<Broadcast, 2000>) and the other to the unique address (<Unicast, 2001>).

Therefore, to support the minimal identification of the members of a multicast set, a single primitive, *bestid* is proposed which, given a multicast set identifier, returns either a multicast set identifier or a unicast identifier, which "best" identifies the Process as a member of the multicast set:

bestid (Identifier, var BestIdentifier)

where:

Identifier: should be a multicast set identifier which has been previously generated by the *newid* primitive;

BestIdentifier: is the "best" identifier which identifies the Process and is returned by the *bestid* primitive.

The value of "BestIdentifier" is clearly dependent upon the underlying Host and Process identification schemes supported by the distributed system. Ideally, the value of the multicast set identifier supplied by *bestid* should be returned. For example, in a distributed system using Host-Port identifiers to identify receiving Processes, the following would be the preferred order of BestIdentifier to be returned:

a) Multicast-Global or Broadcast-Global, potentially allowing the minimum number of transmissions if all possible members return the same Host-Port pair;

b) A Shared Port identifier, which can result in one transmission if all Processes are on different Hosts and Multicast or Broadcast Host identifiers are allowed;

c) A Local Port identifier, which can reduce the number of transmissions to a single Host;

d) A Unique Port identifier, which requires the layer supporting multicast transmission to transmit individual messages to all possible receiving Processes.

The implementation of the *bestid* primitive will be discussed further in Chapter Five.

4.1.1.3.3 Creating a Multicast Set

Once the Source Process has the list of "best" identifiers (supplied by those Destination Processes willing to join the new multicast set), it should be able to associate the "best" identifiers with the multicast set identifier. This is intended to permit the Process to refer to the multicast set identifier, while the layers supporting the multicast transmission would have the list of "actual" identifiers required for the transmission.

To allow this association between the multicast set identifier and the list of "best" identifiers, the following primitive, *create*, is proposed. *Create* binds a textual name (allowing, for example, the use of the *getid* primitive to access a particular multicast set) and the multicast set identifier with the list of "best" identifiers:

ReturnCode := *create* (Name, Identifier, BestList)

where:

Name: is a character string, allowing the identification of the multicast set by textual name, rather than by a numeric identifier;

Identifier: a multicast set identifier;

BestList: a list of (best) identifiers, supplied by the various members of the multicast set;

ReturnCode: an indication as to the success or failure of the *create* primitive (see below).

It is assumed that the *create* primitive searches BestList for duplicate entries and removes them, ensuring the minimum number of identifiers for any transmission.

The ReturnCode simply indicates whether the creation took place ("Success") or whether it failed ("UnableToCreate"). The *create* primitive could fail if, for example, a name server was being used and insufficient space existed for the storage of the information.

Note that *create* need not be restricted to multicast sets. For example, *create* could be supplied with a name to be associated with a single receiving Process, a dummy multicast set identifier and the Process's unicast identifier.

4.1.1.3.4 Deleting a Multicast Set

When a Process no longer requires a multicast set, it should be able to remove the information relating to that multicast set. For example, to free unwanted storage on a name server. Therefore, to complement the *create* primitive, the *remove* primitive is proposed:

ReturnCode := *remove* (Name)

where:

Name: a character string, indicating the name of the multicast set which is to be removed;

ReturnCode: a value indicating the success or failure of the *remove* primitive (see below).

The ReturnCode indicates "Success" if the Name and its associated identifiers were removed. A ReturnCode of "CannotRemove" is returned if, for example, the Name did not exist.

4.1.2 Multicast Communication Primitives

In the second Chapter, two fundamental communication operations were demonstrated in the discussion of multicast applications:

a) the ability to *transmit* a message to one or more Receivers, and

b) the ability to *receive* messages sent by one or more Transmitters.

The next two subsections describe the primitives intended for multicast transmission and multicast reception.

4.1.2.1 Transmission

The transmission primitive should allow the transmitting Process to transmit a message to either a specific receiving Process using a **unicast** identifier, or to a group of receiving Processes using a **multicast set** identifier. In order to avoid having two transmission primitives, one for sending a message to a specific Process (i.e. a unicast transmission) and the other for transmissions to Processes belonging to a multicast set (i.e. a multicast transmission), a single primitive, the *send* primitive, is proposed:

ReturnCode := *send* (Identifier, Message, Size)

where:

Identifier: an identifier identifying the intended Receiver(s) of the message. Since the *send* primitive is to support both unicast and multicast transmission, the identifier must be either a unicast identifier or a multicast set identifier;

Message: the message to be transmitted to the Receiver(s) indicated by the Identifier;

Size: the number of bytes in the message;

ReturnCode: an indication as to the success or failure of the execution of the *send* primitive (see below).

The *send* primitive is to offer a best effort datagram transmission facility only. Since the number of destinations may be unknown, the *send* primitive can only indicate whether the message was sent (with a ReturnCode of "Success"), not whether it was received correctly. However, should the identifier be unrecognized by the *send* primitive or some other failures occur when attempting to perform the transmission, the transmission is aborted and the transmitting Process informed of the failure with a ReturnCode of "NotSent".

Finally, it is assumed that the underlying layers supporting the *send* primitive are responsible for both transmitting the message as well as including an identifier which identifies the intended receiving Process(es) and the unicast identifier of the transmitting Process. This second identifier is necessary to allow receiving Processes to identify the transmitter of the message (should responses be required).

4.1.2.2 Reception

The requirements for reception are somewhat more complex than simply the ability to receive messages destined for a specific receiving Process (using a **unicast** identifier) or group of Processes (using a **multicast set** identifier). For example, in the Time-Server applications described in Chapter Two, only certain time values may have been accurate enough for the Receiver, causing those values deemed inaccurate to be discarded. It was therefore decided that the reception facility should have the ability to **filter** the incoming messages – keeping only those that met certain predefined conditions. In addition, since a single reception could entail receiving *many* messages, as determined by the filter, it was proposed that each received message and the unicast identifier associated with the transmitter of the message should be stored in a linked data structure.

As with the *send* primitive, in order to avoid having several different primitives all essentially performing the same task, a single primitive, the *receive* primitive, is proposed:

ReturnCode := *receive* (Identifier, var MessageList, Filter, TimeLimit)

where:

Identifier: the means by which the receiving Process is identified. Messages received with an identifier matching "Identifier" are to be made available to the Process executing the *receive* primitive. The identifier supplied by the receiving Process can be either a multicast set identifier, indicating the multicast set to which the Process belongs, or a unicast identifier, indicating that the Process is only expecting messages for itself;

MessageList: a list, returned by the *receive* primitive, consisting of any messages which have been received. Exactly which messages are to be kept and the order of storage of the messages in the list is determined by the Filter (see below). If no Filter is supplied, MessageList is to point to a data structure containing the first message received and the unicast identifier of the Transmitter of the message. If a Filter is supplied, the data structure pointed to by MessageList and the number and order of the messages is determined by the Filter;

Filter: a function, supplied by the Receiver, which is used to accept or reject messages based upon some criteria. A null Filter value indicates that no filtering is required and that the first message to be received should be returned in the MessageList. (See below for a further discussion of the Filter);

TimeLimit: the maximum amount of time that the Receiver is willing to wait for a message to be received. When the TimeLimit expires, the Filter function is called (if it has been supplied) with an indication that a time out has occurred. This allows the Filter function to determine the value of the ReturnCode. Should the TimeLimit expire without a supplied Filter function (and therefore no messages were received), *receive* returns an error code of "Timeout" to the calling Process.

An indefinite wait can be caused using a TimeLimit of "-1", while a simple poll, to determine if any messages are available, can be achieved with a TimeLimit of zero;

ReturnCode: an indication from either the *receive* primitive itself or the supplied Filter function as to the success or failure of the receive operation (see below).

Control remains with the *receive* primitive until one of the following events occur (at which point control would be returned to the receiving Process):

a) a message is received and the receiving Process has not supplied a Filter function. This message is made available to the receiving Process through MessageList;

b) a message is received and the Filter detects that a predefined condition has been met (for example, a certain number of messages might have been received or accepted). The number of messages, if any, and their order in the list of messages are determined by the Filter;

c) the TimeLimit has been reached (irrespective of the number of messages accepted). Note that if a Filter is not supplied and an infinite TimeLimit is indicated, control will only return to the receiving Process when a message is received.

The *receive* primitive supplies the Filter function with four parameters: the Transmitter's unicast identifier, the received message, the MessageList pointer, and a time out indicator. Which messages are accepted and placed in the message list depends upon the Filter; however, once the Filter has finished processing the message, it is expected to observe the following rules regarding returning control to the *receive* primitive:

a) a positive return value indicates to the *receive* primitive that the message was accepted;

b) a negative return value indicates to the *receive* primitive that the message was *not* accepted;

c) a return value with an absolute value of "1" indicates that more messages are expected and control is to remain with the *receive* primitive and is not to be returned to the receiving Process;

d) a return value with an absolute value other than "1" indicates that no more messages are expected and control is to be returned to the receiving Process.

When a Filter signals that no more messages are expected, the receive primitive returns the absolute value of the Filter's return value to the receiving Process in the ReturnCode.

In addition to the possible ReturnCode values supplied by the Filter, the *receive* primitive has three predefined ReturnCodes:

-1: indicating that the receive primitive has detected an error. This ReturnCode can occur whether or not a Filter is supplied;

0: a timeout has occurred (without a Filter being supplied);

1: a message has been received (without a Filter being supplied).

4.2 Examples

A transmission (either unicast or multicast) could be implemented using two of the primitives, assuming that the transmitting Process has the name associated with the intended Receiver(s):

```
if getid ("Example1", Example1Id) then
      send (Example1Id, Message, sizeof(Message))
else
      error("Identifier 'Example1' is unknown");
```

In this example, "N" bytes of "Message" are sent to the receiving Process(es) identified by the identifier "Example1Id". Had the identifier "Example1Id" been previously available, there would have been no need to invoke *getid.* The number of Processes actually receiving a copy of the message depends, in part, upon the type of the identifier (either unicast or multicast set) and whether the message reaches the intended Process(es).

The *receive* primitive can be used in several different ways to allow a Process to receive messages. In the following example, the receiving Process is not a member of a multicast set and is to receive messages associated with its own unicast identifier. The first message to be received is returned to the Process, since the timeout value indicates an indefinite wait and no filter is supplied:

```
    getid ("", MyIdentifier);
    MessageList := NIL;
    receive (MyIdentifier, MessageList, NoFilter, Indefinite);
```

As in the previous example, *getid* need only be called once, to establish which identifier is to be used by *receive*. Note that this example is somewhat spurious since the identifier supplied by *getid* may not be known to any transmitting Processes. In an actual application, the Process, prior to receiving, would probably distribute its identifier to potential Transmitters (using, for example, the *send* primitive).

In the following example, a filter function is required to inspect each message that is received for the string "Time-Please". When this string is found, control is returned to the time-server (see below) by returning the value "-2" to the *receive* primitive. Invalid messages are to be discarded, indicated to the *receive* primitive by a return value of "-1":

```
function Filter(UnicastId, Message, var SourceId, TimeOut) : integer;
begin
    if Message = "Time-Please" then
    begin
        new(SourceId);
        SourceId . Requester := UnicastId;
        Filter := -2        { Discard Message and return }
    end
    else
        Filter := -1        { Discard Message and continue receiving }
end;
```

Note that the message itself is not kept, instead, the transmitter's unicast identifier (UnicastId) is returned to the time-server in the variable SourceId.

The time-server consists of a loop in which it waits (indefinitely) for a valid request to have been received; valid requests cause the filter function to return a value of "-2", to which the time-server responds to the requester of the time with the current time value. Should the *receive* primitive fail for some reason, an error message is generated and the time-server *leave*s the Time Service multicast set:

```
getid ("Time-Service", TimeServiceGroup);
join (TimeServiceGroup);
while receive (TimeServiceGroup, SourceId, Filter, Indefinite) = -2 do
begin
    { Obtain time from local clock }
    send (SourceId . Requester, CurrentTime, sizeof(CurrentTime));
    dispose(SourceId)
end;
error("Time-Service stopped");
leave (TimeServiceGroup);
```

The Process requesting the time from the time servers would also require the multicast set identifier of the time servers. Note however, that the Process would not be required to join the time-server multicast set since it is only requesting the time. In the following example, the Process requesting the time accepts the first time value received within five seconds and ignores the rest:

```
getid ("Time-Service", TimeServers);
getid ("", Self);
Message := "Time-Please";
send (TimeServers, Message, sizeof(Message));
if receive (Self, SuppliedTime, NoFilter, FiveSeconds) = Timeout then
      write("No response from Time Servers")
else
      write("Time is: ", SuppliedTime);
```

In the next example, the multicast communication primitives are used to develop an application which is to create a new multicast set consisting of one or more members. The following assumptions will be made regarding this application:

a) there is a Process requiring the generation of a new multicast set, and

b) there is an existing multicast set consisting of one or more Processes which allows its members to join new multicast sets.

The Process requiring the new multicast set must first generate a unique multicast set identifier by which the new multicast set will be identified, and then transmit this identifier to the potential new members with a request that they join this new multicast set. The *receive* primitive is called with a filter which is to create a list of new members pointed to by NewMemberList – if no members are found, NewMemberList will remain NIL after the call to *receive*. Only after the list of identifiers associated with new members has been created can the Process start to transmit messages to the new multicast set:

```
getid ("PotentialMembersSet", PotentialId);
getid ("", MyId);
newid (NewMulticastId);

Request . Type := CanYouJoin;
Request . NewId := NewMulticastId;
send (PotentialId, Request, sizeof(Request));

NewMemberList := NIL;

receive (MyId, NewMemberList, Filter, FiveSeconds);

if NewMemberList = NIL then
    error("No members for new group")
else
begin
    create ("New-Set", NewMulticastId, NewMemberList);
    send (NewMulticastId, Confirmation, sizeof(Confirmation));
    { Send and Receive subsequent Messages }
end;
```

Responses received within the first five seconds are included in the list of new members.

The filter required by the Process is to create a list of the "best" identifiers supplied by the responding new members:

```
function Filter (XmitId; Message; NewMemberList; TimeOut) : integer;
begin
if NOT TimeOut then
begin
    if Message . Type = YesICan then
    begin
        new (NewMember);
        NewMember . Id := Message . BestId;
        NewMember . Next := NewMemberList;
        NewMemberList := NewMember;
        Filter := -1;      { Dispose of message but keep receiving }
    end
    else
        Filter := -1;      { Wrong message type – ignore }
end
else
    Filter := 2;           { Timeout occurred – all done }
end;
```

As each new member's best identifier is received, indicated by the message type of "YesICan", it is included in a list of identifiers, the first element of which is pointed to by NewMemberList.

Before a Process can receive requests to join a new multicast set, it must first join the "PotentialMembers" multicast set. Once joined, it can wait indefinitely for requests of type "CanYouJoin" (handled in this example by a filter). The Process must now respond to the requests with a "YesICan" type message and its "best" identifier. After joining the new multicast set, the Process waits ten seconds for a confirmation – if one is not received, it leaves the new multicast set, otherwise it waits for subsequent messages sent to the (new) multicast set:

```
getid ("PotentialMembers", PotentialMember);
join (PotentialMember);
if receive (PotentialMember, NewList, Filter, -1) = 2 then
begin
    NewMulticastId := NewList . NewId;
    Response . Type := YesICan;
    bestid (NewList . NewId, Response . BestId);
    send (NewList . Requester, Response, sizeof(Response));
    join (NewMulticastId);
    if receive (NewMulticastId, Reply, NULL, TenSeconds) < > -1 then
    begin
        if Reply . Answer = Yes then
        begin
            leave (PotentialMember);
            { Wait for subsequent messages }
        end;
    end;
    leave (NewMulticastId);
end;
leave (PotentialMember);
```

It is worth noting that if this algorithm were to be used in an actual distributed system, the PotentialMember multicast set would rapidly disappear after the first few requests to join new multicast sets. Therefore, in trial implementations, the above algorithm was modified so that it spawned another Process upon receipt of a "CanYouJoin" message:

```
getid ("PotentialMembers", PotentialMember);
join (PotentialMember);
repeat
      rc := receive (PotentialMember, NewList, Filter, -1);
      if rc = 2 then
      begin
            { Spawn new Process }
            { Perform "joining new multicast set " (as described above) }
      end
until
rc <> 2;
error("Potential Member error");
leave (PotentialMembers);
```

This ensures that each Host which supports the PotentialMembers multicast set can continuously set up new multicast sets.

4.3 Related Work

Although in previous Chapters other multicast communication implementations have been described, only one, the V-System developed at Stanford, has been published with a detailed description of the available primitives. Therefore in this Section, the primitives associated with the V-System are compared with the multicast communication primitives that have been proposed in this Chapter.

The V-System supports multicast communications using a Broadcast-Global identifier scheme (i.e. all Processes which are members of a multicast set share a Global identifier and all Hosts on the Ethernet receive a copy of every message sent, irrespective of whether they support members of the set or not).

The V-System implements a Client-Server model of communications. That is, a Client Process is initially a Transmitter, transmitting a multicast message to the Server Process(es), acting as Receivers. Once the message has been processed, the Servers become transmitters, while the Client Process becomes a Receiver.

The V-System supports a total of ten primitives [Cheriton1985a]:

CreateGroup: creates a multicast set in which the Process creating the multicast set becomes the first member;

JoinGroup: allows a Server to join an existing multicast set;

LeaveGroup: allows a Server to leave a multicast set of which it is currently a member;

QueryGroup: allows a Process to determine the membership of a multicast set;

Send: used by the Client to send a Request to a multicast set;

Reply: used by a Server to reply to a Client;

ReceiveSpecific: allows a Process to receive Requests destined to a specific address;

GetReply: used by a Client to obtain a reply from the queue of replies maintained by the kernel;

DestroyProcess: allows a member of a multicast set to destroy any other member of the multicast set (there are certain access privileges required to perform this primitive);

ForceException: allows a Client to send an exception condition (i.e. a break signal) to the members of the multicast set.

There now follows a brief comparison of the V-System's multicast communication primitives with those proposed in this Chapter.

4.3.1 Send and Reply

Unlike the V-System, the proposed primitives make no distinction between a "Client *send*" and a "Server *reply*", treating both as transmissions requiring the proposed *send* primitive.

There is at least one argument supporting the inclusion of the *reply* primitive – it allows the kernel to recycle much of the information and storage associated with the original message, thereby reducing communication times. For example, the Server does not need to supply the address of the Client since the kernel has access to it, nor does the kernel have to reformat an Ethernet packet, since the original packet is still available.

4.3.2 ReceiveSpecific and GetReply

The V-System also makes a distinction between a "Client *ReceiveSpecific*" and a "Server *GetReply*", which are simply variations of the proposed *receive* primitive.

For example, if a specific message is required, the *receive* primitive allows the Process to supply a filter which can discard all those messages not wanted (i.e. a *ReceiveSpecific*). Similarly, the next available message can be obtained by invoking the *receive* primitive without a filter, resulting in the next available message to be returned (i.e. a *GetReply*).

4.3.3 ForceException and DestroyProcess

The proposed primitives do not directly support facilities corresponding to either of the V-System's *ForceException* or *DestroyProcess* primitives. Instead, it was assumed that the Processes using the proposed primitives would implement a remote procedure call facility (using the *send* primitive) which would use existing system calls to support features such as destroying Processes or forcing exceptions.

4.3.4 JoinGroup and LeaveGroup

The *JoinGroup* primitive and the *LeaveGroup* primitive are essentially identical to the proposed *join* and *leave* primitives, respectively.

4.3.5 QueryGroup

The proposed primitives do not directly support the V-System's *QueryGroup* primitive. Instead, a Source Process is expected to use the *send* primitive to transmit an inquiry message, which (assuming the protocol existed) would cause the various Destinations to respond with their unicast identifiers. The Source could either filter the replies to build up a list of unicast identifiers or it could receive each identifier individually and store it in a list.

4.3.6 CreateGroup

Multicast set creation differs quite widely between the proposed primitives and the V-System. The V-System's *CreateGroup* primitive involves the following four steps:

1. generate a random number and use this as the Group Number (i.e. a multicast set identifier);
2. send this Group Number to the network;
3. if replies are received (within an allotted period of time), the Group Number generated was not unique, therefore repeat from step 1;
4. the Group Number is assumed to be unique, and the initiator of the *CreateGroup* primitive now joins the new group.

This approach differs from proposed primitives in two respects:

a) the Group Number is generated randomly and, to ensure its uniqueness, is transmitted - allowing the *CreateGroup* primitive to determine just how unique the Group Number is, and

b) the creator of the group becomes its first member.

CreateGroup can be emulated, more rapidly, using the *newid* and *join* primitives. This approach is more attractive in that it involves no network traffic when generating the new multicast identifier. The proposed primitives also avoid the possibility of having duplicate multicast set identifiers (which are possible in the V-System if Processes which are members of an existing multicast set (i.e. with the same Group Number) do not respond within the allotted period because of problems such as network traffic or machine crashes [Zwaenepoel1985a]).

4.3.7 Summary

The proposed primitives differ from those supported by the V-System in several ways:

a) the filtering capabilities supported by the proposed primitives considerably reduce the amount of unnecessary message handling by a receiving Process;

b) the generation of unique identifiers for multicast set identification (using the *newid* primitive) ensures that members of the multicast set are uniquely identified. This approach seems considerably more flexible than, for example, the method adopted by the V-System in which the Process "randomly" generates multicast set identifiers until a unique one is produced;

c) the proposed primitives support a simple Transmitter-Receiver model rather than, for example, a Client-Server model such as the one used by the V-System. This potentially allows more design flexibility since, as shown in Chapters Two and Three, not all applications are easily described using the Client-Server model and models such as the Client-Server can be built out of the Transmitter-Receiver model.

4.4 Concluding Remarks

In this Chapter, a set of nine multicast communication primitives have been proposed, the design of which was influenced by the various taxonomies introduced in Chapter Three. In addition to proposing the primitives, it was also shown how the primitives could be used by describing some of the applications presented in Chapter Two.

When comparing the proposed primitives with those of the V-System, it is apparent that a great deal of functional commonalty exists (such as transmitting to a set of receivers using a single primitive and the ability to join and leave a multicast set). However, the proposed primitives seem more flexible than those used in the V-System for several reasons:

a) by using the Client-Server model, the V-System does not directly support multicast reception – rather it supports a form of filtered unicast reception;

b) by recognizing that there are different types of identifier possible, the proposed primitives are not necessarily tied to one type of network.

The next two Chapters will demonstrate how the proposed primitives can be implemented on a variety of intranetworks and internetworks.

Chapter 5 Intranetwork Multicast Communications

With the exception of the *bestid* primitive, the primitives developed in Chapter Four were discussed in terms of Processes rather than of specific distributed systems or networks. However, since one of aims of this study of multicast communications is to demonstrate how the primitives could be implemented on a variety of networks and distributed systems, the proposed primitives must be discussed in a wider context.

The purpose of this Chapter is twofold. First, to discuss how different intranetwork architectures affect the implementation of the primitives and second, to compare the transmission and reception overheads of the primitives implemented on these architectures.

By considering an intranetwork as simply a layer, the number of different intranetworks that need examination is reduced to four, since, as the multicast transmission taxonomy demonstrated, there are only four different types of communication possible within a layer (One-Unique, One-Group, Many-Unique, and Many-Group). Therefore, to illustrate the problems associated with implementing the primitives, only four different intranetwork architectures (one from each category) are required.

This Chapter is organized as follows. In the next Section, the facilities available within the Computing Laboratory for the implementation of the multicast communication primitives are presented. Since the available facilities did not support all four possible intranetwork architectures, it was necessary to develop (layered) software to emulate some of the different categories. These different intranetwork communication architectures are described in Section Three. In Section Four, the problems associated with implementing the primitives on each of the different intranetwork architectures are then discussed. In the fifth Section, the performance of the *send* and *receive* primitives in the different implementations are compared. The Chapter is concluded with a series of observations and comments regarding the performance results.

5.1 Facilities

In this Section, the hardware and software facilities available in the Computing Laboratory for the implementation and testing of the multicast communication primitives are described.

5.1.1 Hardware

Six hosts were available for the implementation and testing of the multicast communication primitives. These were (ranging in speed, from fastest to slowest [Parrington1986a]): a SUN-3, an Orion, two VAX-750's, and two Whitechapels. A 10Mb Ethernet interconnected all the machines. The Ethernet interface hardware on each of the Hosts supported a unicast (Host) address and the Ethernet broadcast address (these two addresses being "predefined" by Xerox [DIX1980a]).

5.1.2 Software

All six Hosts used in these experiments supported UNIX (one VAX-750 ran 8th Edition UNIX, while the remaining Hosts ran UNIX Version 4.2). Both versions of UNIX provided **sockets**, a mechanism which permits interprocess communications [Leffler1983a].

Briefly, a socket is a Host-Port identifier bound to a Process. It is a general purpose communication mechanism usually associated with either of the following protocols:

a) UDP, or User Datagram Protocol, which does *not* guarantee the arrival of the message to the intended receiving Process;

b) TCP, or Transmission Control Protocol, which ensures that messages arrive at the receiving Process in the order they are sent without loss or duplication.

Normally only one Process can access a socket. The one exception to this rule is that any Child Process spawned by a Parent Process can share access to the Parent's socket.

In an interprocess communication (or IPC), the transmitting Process supplies its **Socket Layer** with a message and the Host-Port identifier to which the message is addressed. The Host identifier is used by the Socket Layer in determining the routing of the message; this usually involves mapping the Host identifier (or **Internet Address**) into the physical Destination (Host) address supported by the underlying network. The Host-Port identifier and the message are then transmitted using whatever packet structure and protocol are required by the underlying network to the Host indicated by the Destination address. For example, a socket transmission on the Ethernet involves building an Ethernet packet consisting of the Source and Destination Host-Port identifiers and the message. The packet is then transmitted with the Destination Host's Ethernet address (which is not necessarily the same as the Destination's Internet Address).

Upon receipt of a message, the Ethernet interface supplies the message to the receiving Socket Layer on the Destination Host. The Destination Internet Address from the message is then examined by the Socket Layer. If the Destination Host Internet Address from the message corresponds to that of the receiving Host, the message is kept. The Port identifier (from the message) is then used to determine which Process (if any) the message is to be supplied to.

For example, consider the following scenario, using a simplified version of UDP, in which Process-1 on Host-1 is to send a message addressed to Port 4321 on Host-2:

1. Process-1 supplies a message for <"Host-2", 4321> to Host-1's Socket Layer.	*1.* Process-2 binds to <"Host-2", 4321>.
2. Host-1's Socket Layer maps "Host-2" into Host-2's Ethernet Address.	*2.* Process-2 waits for any message to arrive at <"Host-2", 4321>.
3. Host-1's Socket Layer sends the message and <"Host-2", 4321> with Host-2's Ethernet Address.	
	3. Host-2's Ethernet hardware recognizes its Ethernet Address in the packet and takes a copy.
	4. Host-2's Socket Layer recognizes the Internet Address as its own and keeps the message.
	5. Host-2's Socket Layer supplies the message to Process-2, bound to Port 4321.

Figure 5-1: Socket Communication

Port identifiers can either be assigned dynamically to a Process by the Socket Layer itself or the Process can request a specific Port identifier. A message arriving at a Socket Layer with a Port identifier which is not currently associated with a Process is discarded by the Socket Layer.

Clearly, from this discussion of sockets, one can conclude that sockets offer One-Unique transmission at the Network Layer (i.e. Host-to-Host) and One-Unique transmission at the Socket Layer (i.e. Process-to-Process). It is possible however to obtain One-Group transmission at the Network Layer by setting the Host identifier to a well known (predefined) broadcast value and to transmit the message using the UDP protocol. Messages sent with the broadcast (Host) identifier are received by *all* Hosts on the network. As in the case of One-Unique Network Layer transmissions, the message is made available to the Process associated with the Port indicated in the Host-Port identifier. Now however, instead of a single Process on a specific Host receiving a copy of the message, *all* Processes on *all* Hosts which have access to the specified Port receive a copy of the message. For example, if Process-1 on Host-1 in Figure 5-1 (above) had transmitted its message to <"All", 4321>, *all* Processes associated with Port 4321 on *all* Hosts would have received a copy of the message. (This technique is used by several socket utility programs such as **rwho**, for the distribution of network information to well known system Ports [Linton1986a].)

Interprocess communication within a Host is also possible using sockets. For example, if Process-1 on Host-1 in Figure 5-1 had transmitted its message to <"Host-1", 4321> (instead of "Host-2"), the Process associated with Port 4321 on Host-1 would have received a copy of the message. On UNIX 4.2 Hosts, socket transmissions between Processes on the same Host do not proceed onto the network, instead they remain on the Host in question.

5.2 Intranetwork Architectures

The communication facilities offered by sockets conform to the Host-Port model described in Chapters Two and Three. That is, for a Process to communicate with another Process, it must supply the Communication Layer (i.e. the Socket Layer) with an identifier consisting of two parts - a Host number and a Port number (to which the receiving Process is expected to associate itself).

The basic socket facilities allow, at best, a One-Unique type of multicast transmission (i.e. both the Host and Port identifiers indicate unique destinations). Therefore, a multicast transmission using the basic socket facilities would require that the message be sent repeatedly, a copy to each possible receiving Process.

Fortunately, as indicated earlier, sockets do support a form of One-Group multicast transmission, in that the Host identifier can be set to indicate *all* Hosts and then be mapped by the Socket Layer into the Ethernet's broadcast identifier. A transmitting Process can therefore send a message to all Hosts. However, as before, only a single Port can be identified.

As it stands, only a restricted form of One-Group multicast transmission can be implemented (in addition to One-Unique multicast transmission), since sockets allow at most one receiving Process on a single Host. Consequently, to examine the implementation of the primitives on One-Group (allowing multiple receiving Processes on a single Host), Many-Unique, and Many-Group intranetwork architectures, the following design alternatives were considered:

a) to modify (or replace) the existing socket software so that all four of the intranetwork architectures could be supported, or

b) to add a layer of software between the communicating Processes and the Socket Layer. This new layer would then emulate the required type of multicast transmission using the underlying socket interface.

Of these alternatives, the first (replacing or modifying the existing socket software) was rejected for three reasons. First, it was assumed that the effort associated with changing the existing socket software would be too time consuming. Second, by creating our own version of sockets it was probable that the new socket software would be incompatible with the socket software at other UNIX sites. Third, some of the UNIX Hosts were not supplied with source code - making software changes to the sockets all but impossible. Therefore, the remaining alternative, adding a layer of software, was the only possible choice.

The requirements, design and implementation of a series of software layers which require those types of intranetwork architectures not directly supported by sockets (that is, One-Group, Many-Unique, and Many-Group) are now considered.

5.2.1 The Multicast Communication Layer

From the discussion in the previous Section, it is apparent that additional layers of software would be required for each of the intranetwork architectures that could not be directly supported by sockets (i.e. One-Group, Many-Unique, and Many-Group). This new layer, the Multicast Communication Layer, would be responsible for supporting the management of the multicast sets as well as the transmission and distribution of messages on each Host. The Multicast Communication Layer would lie between the Process and Socket

Layers:

Process Layer

Multicast Communication Layer

Socket Layer

Figure 5-2: The Multicast Communication Layer

In order to avoid having the One-Group, Many-Unique and Many-Group intranetwork architectures being emulated by a series of One-Unique (i.e. unicast socket) transmissions, it was decided to:

a) have each receiving Multicast Communication Layer share a common Port, and

b) transmit all multicast messages with a **broadcast** identifier to this common Port.

For testing purposes, it was assumed that a Host could support any of the four multicast communication classifications. However, three receiving Multicast Communication Layers were developed for the distribution of messages on a receiving Host (one each for One-Group, Many-Unique and Many-Group). Each of the Multicast Communication Layers were associated with a separate Port (note, One-Unique did not have a separate receiving Multicast Communication Layer since messages were to be sent directly to the Process in question):

Classification	Port
One-Group	9999
Many-Unique	9989
Many-Group	9979

Figure 5-3: Multicast Layer Port Assignment

For example, socket Port 9999 on all the UNIX hosts supporting multicast communications would be used by the receiving One-Group Multicast Communication Layer.

A multicast transmission would therefore consist of the transmitting Process supplying its transmitting Multicast Communication Layer with a multicast set identifier and the message to be transmitted. The transmitting Multicast Communication Layer would then store the message and its associated identifier(s) (either Unique or Group, see Section 5.2.2) in a socket message and broadcast it to the receiving Multicast Communication Layer under test (as indicated by the Port number). For example, in the One-Group intranetwork architecture, the transmitting Process would supply the One-Group Multicast Communication Layer with a multicast set identifier and a message. The transmitting Multicast Communication Layer would then broadcast the message and the multicast set identifier to Port 9999.

To allow the receiving Multicast Communication Layers to determine which, if any, Processes were to receive a copy of the multicast message, a standard multicast message format was developed for each type of multicast transmission. The Multicast Communication Layer message formats were as follows (note, since the One-Unique intranetwork architecture used sockets directly, no special message format was required):

One-Group:

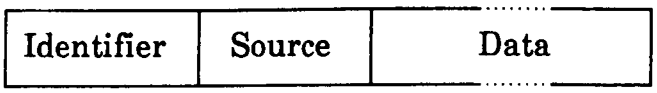

Figure 5-4: One-Group Message Format

where:

Identifier - a Group identifier (see Section 5.22), indicating the multicast set to which the Data is to be distributed to, Source - a Unicast-Unique Host-Port (socket) identifier, identifying the Source of the message,

Data - the information to be transmitted to the members of the multicast set.

Many-Unique:

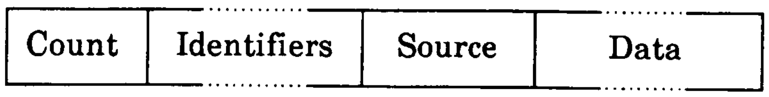

Figure 5-5: Many-Unique Message Format

where:

Count - the number of Unique identifiers associated with the message,

Identifiers - a series of one or more Unique identifiers (each stored as a Unicast-Unique Host-Port identifier, see Section 5.2.2), indicating the intended Receivers,

Source - a Unicast-Unique Host-Port (socket) identifier, identifying the Source of the message,

Data - the information to be transmitted to the members of the multicast set.

Many-Group:

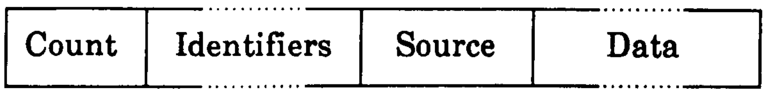

Figure 5-6: Many-Group Message Format

where:

Count - the number of Group identifiers associated with the message,

Identifiers - a series of one or more Group identifiers, indicating the intended Receivers,

Source - a Unicast-Unique Host-Port (socket) identifier, identifying the Source of the message, and

Data - the information to be transmitted to the members of the multicast set.

Upon receipt of a multicast message, the receiving Multicast Communication Layer would examine the identifier(s) associated with the multicast message and determine which Processes on its Host, if any, were to receive a copy of the message. Message distribution by the receiving Multicast Communication Layer was done by performing a series of One-Unique (Process-to-Process) socket transmissions on the Host in question. For example, a Many-Unique Multicast Communications Layer would examine each Host-Port pair, from the multicast message, to determine if:

a) the supplied Host identifier matched that of the receiving Host, and

b) a Process associated with the supplied Port identifier existed on the receiving Host.

Only if both the Host and Port identifiers supplied with the message matched a Host-Port pair maintained by the receiving Multicast Communication Layer would the message be distributed to the destination Process (associated with the Host-Port pair).

5.2.2 Identifier Structures

In addition to the different Multicast Communication Layers and their associated packet structures, it was also necessary to develop identifier structures suitable for each of the Multicast Communication Layers.

Each of the Multicast Communication Layers were influenced by the type of identifier used by the surrounding layers. For example, to support the Process Layer, the Multicast Communication Layer had to allow both multicast set identifiers and unicast identifiers, while when dealing with the Socket Layer, the Multicast Communication Layer used Host-Port socket identifiers.

5.2.2.1 Transmission Considerations

In a unicast transmission (by a Process using any of the architectures), the message was passed directly to the Socket Layer by the Multicast Communication Layer with the Host-Port identifier supplied by the Process.

The Socket Layer was then expected to transmit the message to the specified Destination indicated by the supplied Host-Port pair by performing a standard UDP transmission.

However, multicast transmission (by a Process on any of the architectures) required that the transmitting Multicast Communication Layer map the supplied multicast set identifier into a list of either Unique identifiers or Group identifiers. Then, depending upon the type of transmission under consideration, it had to transmit the message to the intended Destinations, either by unicasting the message (i.e. when considering One-Unique intranetworks) or by broadcasting the message (i.e. when considering any of One-Group, Many-Unique, or Many-Group intranetworks).

5.2.2.2 Reception Considerations

In a unicast reception, the Process expects to receive messages sent with a specific unicast identifier. A multicast reception is similar, with the exception that the Process expects to receive messages sent with a specific multicast set identifier.

5.2.2.3 Identifier Design

The identifier structure decided upon was a simple linked data structure:

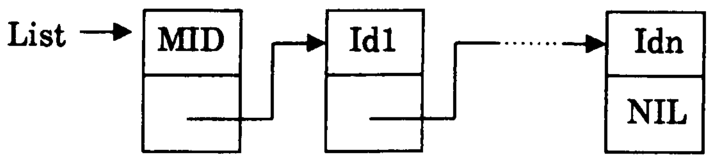

Figure 5-7: Identifier Structure

To distinguish between unicast and multicast set identifiers, the identifier in the first element of the list (MID) could contain either a zero value, indicating a unicast identifier, or a non-zero value, indicating a multicast set identifier.

The subsequent identifiers within the list (Id1 through Idn) were treated by the Multicast Communication Layer as either Unique identifiers or as Group identifiers, depending upon the type of transmission being performed. For example, when performing a transmission with a unicast identifier (i.e. when the MID was zero), Id1 through Idn represented Unique identifiers, while a transmission with a multicast set identifier (i.e. when MID was not equal to zero), Id1 through Idn represented Group identifiers.

Unique identifiers were stored as Unicast-Unique Host-Port (i.e. socket) identifiers and consisted of a 32-bit Host number and a 32-bit Port number. Although the Group identifiers were treated as single 64-bit identifiers, they were stored as pairs of 32-bit integers (a high part and a low part). Id1 through Idn represented a series of either Host-Port identifiers or multicast set identifiers – depending upon the type of network under examination (i.e. Host-Port on Unique intranetworks and multicast set on Group intranetworks).

The multicast set identifier, MID, was also stored as a pair of 32-bit integers. However, how it was used depended upon whether the Process was transmitting or receiving.

For example, when requested to transmit a message with an MID of zero, the transmitting Multicast Communication Layer would treat each identifier in the identifier list as a Host-Port identifier. The supplied message would therefore be transmitted repeatedly to the Process(es) indicated by the Host-Port identifiers. Note, normally an identifier list with an MID of zero would only contain a single Host-Port identifier – indicating a specific Process.

However, if the MID was non-zero, the interpretation of the identifiers and the number of identifiers to be transmitted with each message would be determined by the type of intranetwork architecture under consideration. For example, if the intranetwork architecture was, say, Many-Group, then the transmitting Multicast Communication Layer would transmit a broadcast message to Port 9979 with as many of the (Group) identifiers as could be inserted into the Many-Group multicast message. If the list (Id1 through Idn) contained more identifiers than the message could support, additional messages would be transmitted.

The identifier list was used differently for reception. An MID of zero indicated that the Process was not a member of a multicast set and was expecting a unicast message; the first identifier in the list (i.e. Id1) was taken as the unique Host-Port identifier of the Process. A non-zero MID was taken as the multicast set identifier to which the Process belonged – messages intended for this multicast set (received by the receiving Multicast Communication Layer) would then be supplied to the Process, assuming that it had previously joined the multicast set in question.

5.3 Implementation of the Primitives

To demonstrate that the proposed primitives could operate on any of the four intranetwork architectures, it was decided to implement and test the primitives on each of the architectures. In this Section, the implementation of each primitive is discussed, given the constraints imposed by UNIX sockets, the Ethernet, the Multicast Communication Layers, and the intranetwork architectures.

5.3.1 send

There now follows a description of each of the implementations of the *send* primitive. In each case, it is assumed that a message and a pointer to a list of identifiers were supplied by the transmitting Process.

In the One-Unique intranetwork, both unicast and multicast transmissions could use the same software, which transmitted a copy of the message to each receiving Process as indicated by the list of (Unique) identifiers:

```
Message . Data := DataFromProcess;
Current := List ^. Next;    { Skip MID field }
while Current < > NIL do
begin
    { Transmit the Message to Current ^. UniqueIdentifier }
    Current := Current ^. Next;
end;
```

Note, each Unique identifier represented a Unicast-Unique Host-Port identifier.

In the One-Group intranetwork architecture, the *send* primitive was to support both unicast and multicast transmissions, as required by the definition of the *send* primitive. Therefore, the unicast transmission was identical to that described for the One-Unique network (above) and used if the MID value was zero. The multicast transmission was similar to the One-Unique unicast transmission, with the exception that each identifier was assumed to be a Group identifier and was therefore broadcast to the receiving One-Group Multicast Communication Layer (associated with Port 9999):

```
Message . Data := DataFromProcess;
Current := List ^. Next;    { Skip MID field }
while Current < > NIL do
begin
    Message . Identifier := Current ^. GroupIdentifier;
    { Transmit the Message to <Broadcast, 9999> }
    Current := Current ^. Next;
end;
```

Transmitting messages on a Many-Unique intranetwork, as in the One-Group case, also required two distinct types of transmission - unicast and multicast, as indicated by the value of MID. Unicast transmissions were handled in the same way as One-Unique (above). Multicast transmissions were somewhat more involved because they entailed storing a number of Unique identifiers (i.e. Host-Port identifiers) in the Many-Unique multicast message and then broadcasting the message (and the identifiers) to Port 9989:

```
Message . Data := DataFromProcess;
Current := List ^. Next;    { Skip MID field }
while Current < > NIL do
begin
    I := 0;
    while (Current < > NIL) and (I < > MaxId) do
    begin
        Message . Identifier [I] := Current ^. UniqueIdentifier;
        I := I + 1;
        Current := Current ^. Next;
    end;
    Message . Count := I;
    { Transmit the Message to <Broadcast, 9989> }
end;
```

Transmissions on a Many-Group intranetwork were essentially identical to those on a Many-Unique intranetwork with the exceptions that the identifiers were assumed to be Group identifiers and the message was to be broadcast to Port 9979:

```
Message . Data := DataFromProcess;
Current := List ^. Next;    { Skip MID field }
while Current < > NIL do
begin
    I := 0;
    while (Current < > NIL) and (I < > MaxId) do
    begin
        Message . Identifier [I] := Current ^. GroupIdentifier;
        I := I + 1;
        Current := Current ^. Next;
    end;
    Message . Count := I;
    { Transmit the Message to <Broadcast, 9979> }
end;
```

Note also, that since the GroupIdentifier field and the UniqueIdentifier field have identical structures, it was possible (in theory) to perform a multicast transmission involving a mixture of identifiers (both Unique and Group).

5.3.2 join

The *join* primitive implementation depended upon the type of intranetwork architecture being supported.

Intranetworks supporting Unique identifiers (i.e. One-Unique and Many-Unique) required that the join primitive supply the identifier of the Process to all possible Processes intending to use the multicast set. This was achieved using a simple name server which maintained a list of Host-Port identifiers and the multicast set identifier of the multicast set in question.

In Group intranetworks (i.e. One-Group and Many-Group) as well as in the Many-Unique intranetwork, the *join* primitive supplied the Multicast Communication Layer with a Host-Port identifier of the Process joining the multicast set and the multicast set identifier in question. With this information, the Multicast Communication Layer was able to examine incoming identifiers and distribute the message to the intended members of the multicast set (see Section 5.3.3, below)

5.3.3 receive

The *receive* primitive is discussed in two parts – first, the Multicast Communication Layer, and second, the process of receiving messages.

5.3.3.1 The Receiving Multicast Communication Layer

Before a Process could receive a message sent to a multicast set, the message was first required to pass through the receiving Multicast Communication Layer on the Process's Host (note, this Section does not pertain to the One-Unique intranetwork architectures since One-Unique transmissions were sent directly to the Process associated with the supplied Unique (i.e. Host-Port) identifier).

The function of the receiving Multicast Communication Layer was to distribute the message to the intended destination Processes by examining the identifier(s) supplied with the message. Although the type and number of identifier stored within each message structure could differ (depending upon the intranetwork architecture in question), the basic function of the receiving Multicast Communication Layer was as follows:

```
{ Receive Message from Socket Layer }
repeat
    ExtractIdentifierFromMessage;
    IdPtr := StartOfProcessAndMulticastSetIdentifierList;
    while IdPtr < > NIL do
    begin
        if IdPtr ^. Identifier = ExtractedIdentifier then
            { Send a Copy of the Message to IdPtr ^. Process }
        IdPtr := IdPtr ^. NextIdentifier;
    end;
until
NoMoreIdentifiersInMessage;
```

The process of sending a copy of the message to the intended Process involved a One-Unique socket transmission within the receiving Host.

For example, in a One-Group Multicast Communication Layer, the ExtractedIdentifier would be a Group identifier. Once extracted, the ExtractedIdentifier would then be compared with the Group identifiers (i.e. multicast set identifiers) supplied by the different Processes which had joined the various multicast sets. When an identifier supplied by a Process matched the ExtractedIdentifier, a copy of the message, including the transmitter's

Unique identifier (i.e. Host-Port identifier) would be sent to the Process using a One-Unique socket transmission.

5.3.3.2 Message Reception by the Process

All receptions, both unicast and multicast, used the *receive* primitive. Unicast and multicast reception was distinguished by the value of the MID field within the identifier list (a zero value indicating a unicast reception, while a non-zero value indicated a multicast reception). Unicast reception was implemented as a standard socket reception - allowing messages to be received from any transmitting Process. However, a multicast reception was intended to receive messages only from the Multicast Communication Layer.

Once a message (unicast or multicast) was received, it was stored in a structure to allow the message to be linked to other messages by any filter the Process might supply:

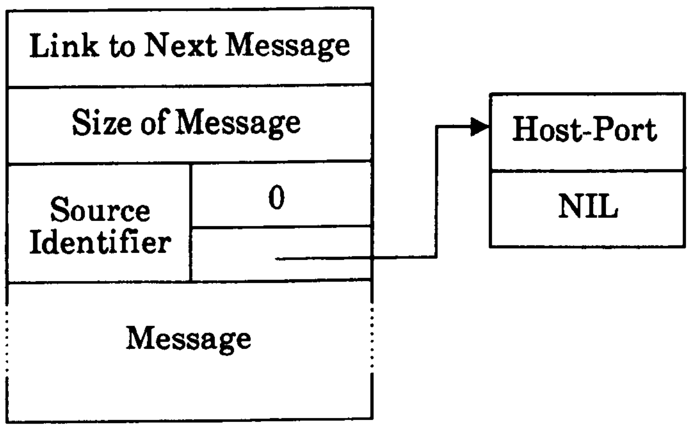

Figure 5-8: Received Message Structure

Note, the Source Identifier was stored in the same format as any other unicast identifier, with an MID of zero (see Section 5.2.2). This was to allow the receiving Process to reply to the transmitting Process by supplying a pointer to the Source Identifier field.

If no filter was supplied, the received message (stored in the structure shown in Figure 5-8) would be returned directly to the receiving Process. However, if a filter *was* available, the message would be supplied to the filter for additional processing. Depending upon the return code returned by the filter to the *receive* primitive, the message would either be kept or discarded and the *receive* primitive would either continue waiting for additional messages or would return the list of messages to the receiving Process.

Since the received message structure now contained both the message and the transmitting Process's unicast identifier, the number of parameters supplied to the filter function was reduced to:

a) the message structure described above (i.e. the message, its size and the Source identifier);
b) the message list (to which the messages could be linked into);
c) a timeout indicator.

5.3.4 leave

The *leave* primitive implementation, like that of the *join* primitive, depended upon the type of intranetwork architecture in use. When leaving a multicast set on a Unique intranetwork, the *leave* primitive informed the name server; which removed the Process's Host-Port identifier from the list of identifiers associated with the multicast set, thereby stopping any new users of the multicast set from sending messages to the Process.

On Group intranetworks, it was only necessary to inform the Multicast Communication Layer that the Process was leaving the multicast set. In addition, in the Many-Unique implementation, the Multicast Communication Layer was also informed, thereby ensuring that messages arriving with the "old" identifier would be ignored and not transmitted to the receiving Process. In both of these cases, the Multicast Communication Layer removed the socket identifier and the multicast set identifier associated with the Process from the list of multicast set identifiers that it maintained.

5.3.5 getid

The *getid* primitive implementation was common to all four types of network in that it accepted a name and returned a list of identifiers. Basically, *getid* accessed a name server which returned the most up-to-date list of identifiers associated with the multicast set. The list of identifiers returned by the name server (consisting of a multicast set identifier, MID, and a list of one or more Unique (i.e. Host-Port) identifiers or Group (i.e. multicast set) identifiers), were formatted by the *getid* primitive into the identifier list structure expected by the Process (see Section 5.2.2).

5.3.6 newid

The *newid* primitive returned a unique 64-bit value constructed from the Process's Host identifier, its Process number and the current time of day (expressed in milliseconds). As with the *getid* primitive, the primitive did not require special implementations on any of the networks.

This method of constructing the identifier was found to be unique in that:

a) Processes on *different* Hosts would not produce the same Host identifier, since within the context of the network, all Host identifiers were assumed to be unique;

b) Processes on the *same* (UNIX) Host always have a unique Process number (at least for the duration of their existence). It was assumed that there would be a significant delay between freeing a Process number and the reissuing of it in the generation of a new multicast set identifier;

c) the time of day value was assumed to be always increasing, even after a machine crash.

However, a Process could get the same value if *newid* was called within the same clock period. Therefore, to avoid a Process obtaining the same value twice, a short time delay was introduced into the *newid* primitive after the generation of a new identifier. This ensured that different multicast set identifiers would be produced after each call to *newid*.

5.3.7 create

The *create* primitive was implemented as described in Chapter Four. That is, given a textual name and a list of identifiers, it associated the name with the list of identifiers. As in the case of *getid*, *create* accessed a name server, supplying it with the textual name of the multicast set, the multicast set identifier and the list of identifiers to be associated with the multicast set.

The *create* primitive accepted three parameters: a name, a 64-bit multicast set identifier, and a linked list of identifiers (Unique or Group), as described in Section 5.2.2.3.

As with the two previous primitives, the implementation of the *create* primitive was common to all four intranetwork architectures.

5.3.8 remove

The *remove* primitive was implemented as described in Chapter Four - given a textual name associated with an identifier, *remove* attempted to remove the name from the list of identifier names. As with *create* and *getid*, the supplied name was supplied to a name server, which removed the information associated with the name.

The implementation of the *remove* primitive was common to all four intranetwork architectures.

5.3.9 bestid

The *bestid* primitive returned a 64-bit identifier, the type of which depended upon the type of network being examined.

In the case of Unique intranetworks, the unique Host-Port identifier associated with the Process was returned as the "bestid". However, in the case of Group intranetworks, the multicast set identifier supplied by the Process was returned as its "best" identifier.

5.4 Performance Results

After all nine primitives were successfully implemented and tested on the four different intranetwork architectures, it was decided to contrast the observed performances of the implementations with those expected from the multicast transmission taxonomy in Chapter Three.

To this end, three tests were devised to compare:

a) the overheads associated with using the Multicast Communication Layer;

b) the costs of distributing a multicast message to Processes residing on a single Host;

c) the costs of distributing a multicast message to Processes residing on different Hosts.

All the tests were fundamentally the same, consisting of a Source Process transmitting a message (of 4 or 512 bytes in length) and receiving a 4-byte response from one or more Destination Processes (determined by the number of Destinations being tested). One thousand of these message cycles (Request-Response) were timed and recorded for subsequent analysis.

The Source Process used in these tests consisted of two parts, a Source procedure which transmitted the message, recording the total time taken for the round trip and a filter function, which received messages from the responding Destinations until all the Destinations had been heard from (or a time limit was exceeded).

The filter function was written as follows:

```
var
Responded, Total, MsgNo : integer;           { Global - see below }

function Check(Response, List : MsgPtr; Timeout : Boolean);
begin
if Timeout then
      Check := -3              { Failure indication - stop processing }
else
if Response ^. SeqNo = MsgNo then
begin
      Responded := Responded + 1;
      if Responded = Total then
            Check := -2        { Success indication - stop processing }
      else
            Check := -1;       { Don't keep response but keep receiving }
end
else
      Check := -1;             { Wrong sequence number - keep trying }
end;
```

The Check function was called by the *receive* primitive each time a message was received. If the sequence number in the Response (SeqNo) agreed with the expected sequence number (MsgNo), the number of responses (Responded) was increased and if the number of responses were found to equal the number of Processes under test (Total), control was returned to the calling Process (with a return code of "-2", indicating successful completion), otherwise the *receive* function was to keep receiving. If the wrong sequence number was detected, the message was discarded (return code "-1") and the *receive* routine continued waiting for messages. Finally, should a timeout occur, control was returned to the calling sequence with an indication that not all messages were received in the allotted time (return code "-3").

The Source procedure, below, recorded the total time required to send a message to the members of the multicast set and to receive responses from all of them. Each message was sent with a sequence number (MsgNo) which was to be returned by each responding Destination:

```
procedure Source(NumberInSet, MsgSize : integer; TestGroup : string);
var
int rc;
Message : TestRecord;
Elapsed : real;
GroupId, Self : IdentifierType;
begin
getid ("", Self);
getid (TestGroup, GroupId);
Total := NumberInSet;
for MsgNo := 1 to 1000 do
begin
    Message . SeqNo := MsgNo;
    Responded := 0;
    StartClock;
    send (GroupId, Message, MsgSize);
    rc := receive (Self, StartOfList, Check, FiveSeconds);
    Elapsed := StopClock;
    if rc = Success then
        write("Success", Elapsed)
    else
        write("Failure", Elapsed)
end;
end;
```

In the tests performed and described in this Chapter, the clock was found to have an accuracy of plus or minus 10 milliseconds [UNIX1983a].

The Destination Process was essentially an echo facility, echoing the sequence number it received from the Source Process. The Destination Process first joined a specific multicast set and then waited for the arrival of a message. Upon receipt of a message, the sequence number was returned to the Source:

```
procedure Destination(TestGroup : string);
var
rc : integer;
GroupId : IdentifierType;
Message : MsgPtr;
begin
getid (TestGroup, GroupId);
join (GroupId);
repeat
    rc := receive (GroupId, Message, NoFilter, WaitForever);
    if rc = 2 then
        send (Message ^. Source, Message ^. SeqNo, FourBytes);
    dispose(Message);
until
rc < > 2;
leave (GroupId);
end;
```

5.4.1 Multicast Set Membership Representation

In each of the performance tests, the number of members of any one multicast set (i.e. the number of Destination Processes) was varied from one to five (five was chosen as the upper limit since there were only six hosts available for testing, one of which had to support a Source Process – the remaining five Hosts were therefore available to support Destination Processes).

For the purposes of the performance tests, the number of Unique or Group identifiers allowed with each message, the type (i.e. Unique or Group) of each identifier, and the number of identifiers required to represent all five members of the multicast set on each of the different intranetwork architectures were as follows:

Intranetwork Architecture	Identifiers per message	Number of Identifiers required to represent all members	Comments
One-Unique	1	5	Each Process is identified by a single Unique (Host-Port) identifier.
Many-Unique	4	5	Each Process is identified by a single Unique (Host-Port) identifier.
One-Group	1	1	All Processes share a single Group (Multicast Set) identifier.
Many-Group	4	5	Each Process is identified by a separate Group (Multicast Set) identifier.

Figure 5-9: Multicast Set Membership Representation

The maximum number of identifiers per message (i.e. "many") was set at four to allow an examination of the effect of multiple message transmission to multicast sets on Many-Unique and Many-Group intranetwork architectures. To achieve this on the Many-Group intranetwork architecture, each member of the multicast set was represented by its own multicast set identifier (i.e. the multicast set used by the Source Process was created out of other, existing multicast sets).

All members of the multicast set on the One-Group intranetwork architecture were represented by a single multicast set identifier (irrespective of the number of members). This helped establish a lower limit on the transmission time required by those intranetwork architectures not directly supported by sockets (i.e. One-Group, Many-Unique, and Many-Group).

5.4.2 Multicast Communication Layer Overheads

The object of this test was to determine the overheads associated with the different Multicast Communication Layers compared to that of performing a One-Unique transmission using sockets directly. The test consisted of sending a four byte message to a *single* Destination Process using whatever method of multicast transmission the specific intranetwork architecture supported. A four byte message was chosen for two reasons: first it represented a small percentage of the overall socket message size, and second, it was the size of the integer sequence number associated with each message. The Destination Process returned the same four byte message as a response using a unicast transmission. (Note that all Multicast Communication Layers performed unicast transmissions using a UDP socket transmission directly.)

Two tests were performed. The first was conducted between a pair of Whitechapels, the slowest of the available UNIX machines, while the second consisted of a Whitechapel sending messages to a SUN-3, the fastest of the UNIX machines.

The tables in Figure 5-10 show the results of the tests. The round trip time is expressed in milliseconds, while the overheads (that is, the difference between the round trip time and the One-Unique round trip time) are expressed as a percentage of the overall round trip time. Note that in addition to the four intranetwork architectures discussed in this Chapter, Many-Unique and Many-Group transmissions were tested in two different configurations:

a) Many-Unique-A and Many-Group-A: consisted of transmitting the message with a single identifier (Unique or Group), that of the intended Destination Process, and

b) Many-Unique-B and Many-Group-B: consisted of transmitting the message with the maximum number of identifiers (four) and the intended

	Intranetwork Type					
	One-Unique	Many-Unique (A)	Many-Unique (B)	One-Group	Many-Group (A)	Many-Group (B)
Round Trip (milliseconds)	60	104	105	102	102	103
Overhead	0	42%	43%	41%	41%	42%

Whitechapel to Whitechapel

	Intranetwork Type					
	One-Unique	Many-Unique (A)	Many-Unique (B)	One-Group	Many-Group (A)	Many-Group (B)
Round Trip (milliseconds)	36	41	43	40	42	42
Overhead	0	12%	16%	10%	14%	14%

Whitechapel to SUN-3

Figure 5-10: Multicast Communication Layer Overheads

Destination Process's identifier stored as the last identifier in the list – thereby requiring the receiving Multicast Communication Layer to scan the entire list of identifiers.

If the obvious speed differences between the machines are ignored, one finds, not surprisingly, that the One-Unique architecture using sockets directly is the fastest, since there is no overhead associated with the handling and distribution of messages by the receiving Multicast Communication Layer. The communication times associated with Many-Unique-A, One-Group, and Many-Group-A are similar because these messages only contained a single identifier, requiring the receiving Multicast Communication Layer to perform the minimum amount of processing. Similarly, the overheads associated with Many-Unique-B and Many-Group-B are slightly higher than Many-Unique-A and Many-Unique-B because of the additional processing required to scan the entire list of identifiers in the message.

From these results, one can conclude that multicast transmissions to the Whitechapels are limited by the rate at which messages can be distributed on the machine, rather than the speed of the Ethernet. On the other hand, because of the speed of the SUN, the overheads associated with any of the Multicast Communication Layer implementations was small compared to the overall communication time.

The large difference in communication times between the Whitechapel and the SUN may not have been entirely due to machine speeds. For example, it is known that the socket implementation on the SUN has been designed to operate with the minimum of overheads. Similarly, the time required for context switching (i.e. changing from User space to Kernel space) on the Whitechapel is quite large, which may also explain some of the overheads associated with performing message distribution.

5.4.3 Destinations on a Single Host

In the following set of tests, the overheads associated with the Multicast Communication Layer distributing messages to the members of a multicast set all residing on a single Host are examined. The tests involved transmitting a multicast message to a multicast set consisting of an increasing number of receiving Processes and determining the round trip time of the message. All four intranetwork architectures were examined (One-Unique, One-Group, Many-Unique, and Many-Group).

The timings were taken for a varying number of destination Processes (starting at one and increasing to five) and different message sizes (4-byte and 512-byte). Since the maximum number of identifiers allowed with the "many" identifier packet was four, the effect of the number of receivers exceeding the number of identifiers per message was also examined.

In Figures 5-11 and 5-12, transmissions between a VAX-750 and a SUN-3 are presented, illustrating the round trip transmission times for 4-byte and 512-byte messages respectively. In Figure 5-13, the various transmission speeds are shown for a VAX-750 transmitting a 4-byte message to a Whitechapel. The following observations are made from these graphs:

a) not surprisingly, the speed of the machine affects the overall message distribution time;

b) if the Process-to-Process transmission time on the network is less than the Process-to-Process transmission time on a Host, it can be more efficient to use a One-Unique multicast transmission (i.e. using the network) rather than having the Destination Host distribute the messages;

c) if the Process-to-Process transmission time on the network is greater than the Process-to-Process transmission time on a Host, it can be more efficient to have the Destination Host perform the distribution of the messages (i.e. using the Multicast Communication Layer) rather than have the Source perform the distribution;

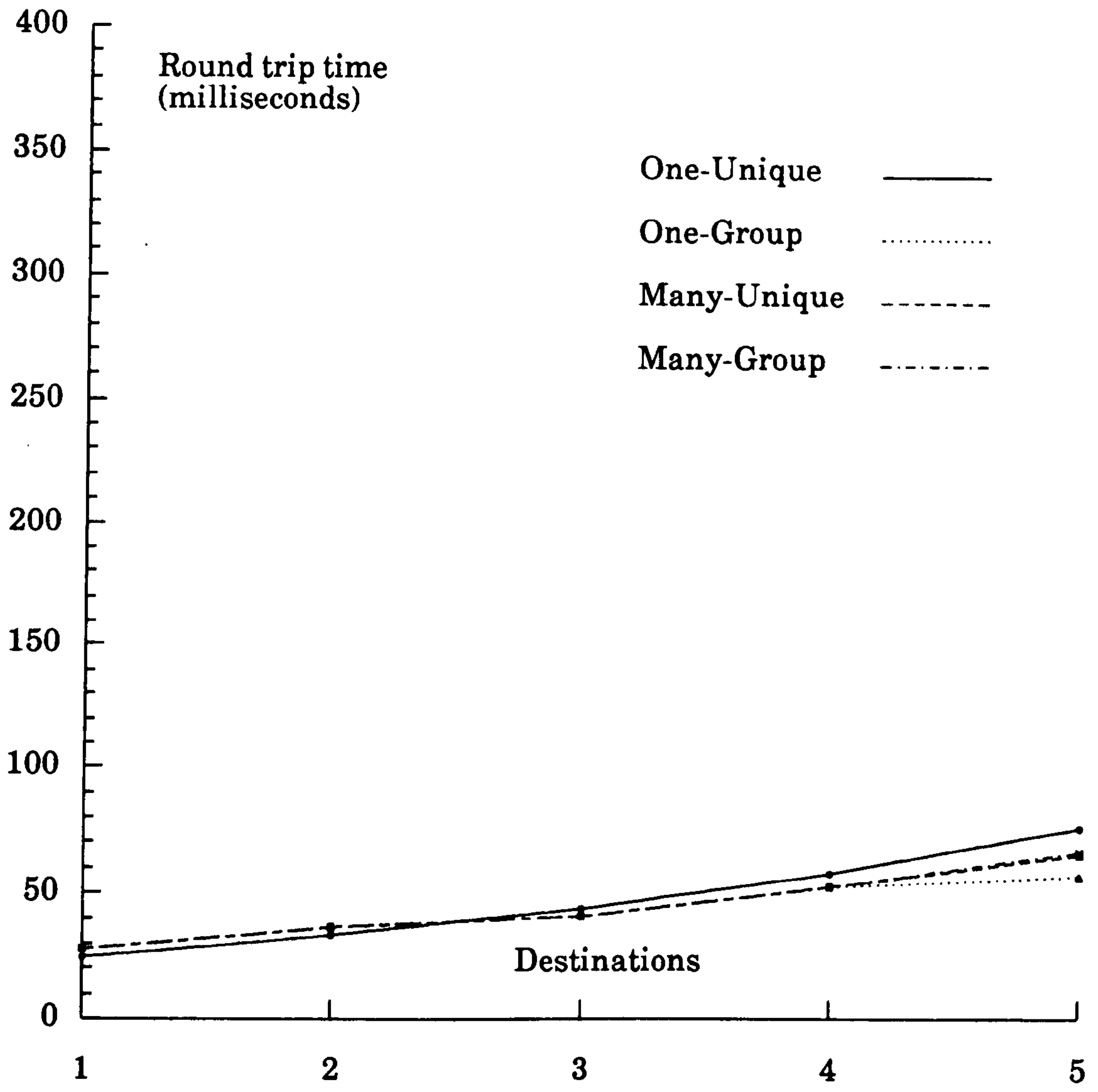

Figure 5-11: VAX-750 to SUN-3 (4 byte message)

Notes:

a) Many-Unique and Many-Group transmissions allowed at most four identifiers to be associated with a single transmission, hence the increased slope when the number of destinations is increased to five.

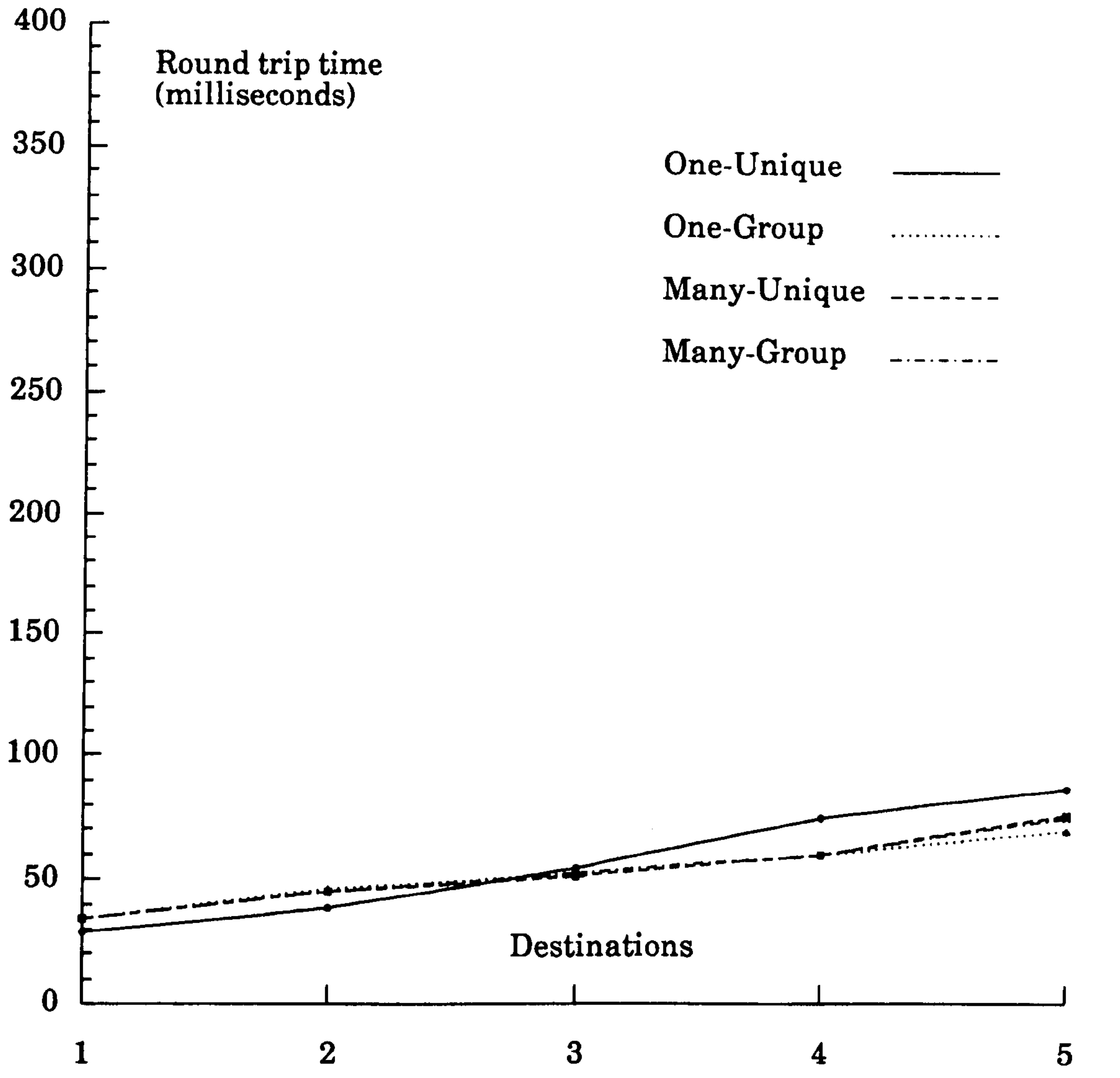

Figure 5-12: VAX-750 to SUN-3 (512 byte message)

Notes:

a) Many-Unique and Many-Group transmissions allowed at most four identifiers to be associated with a single transmission, hence the increased slope when the number of destinations is increased to five.

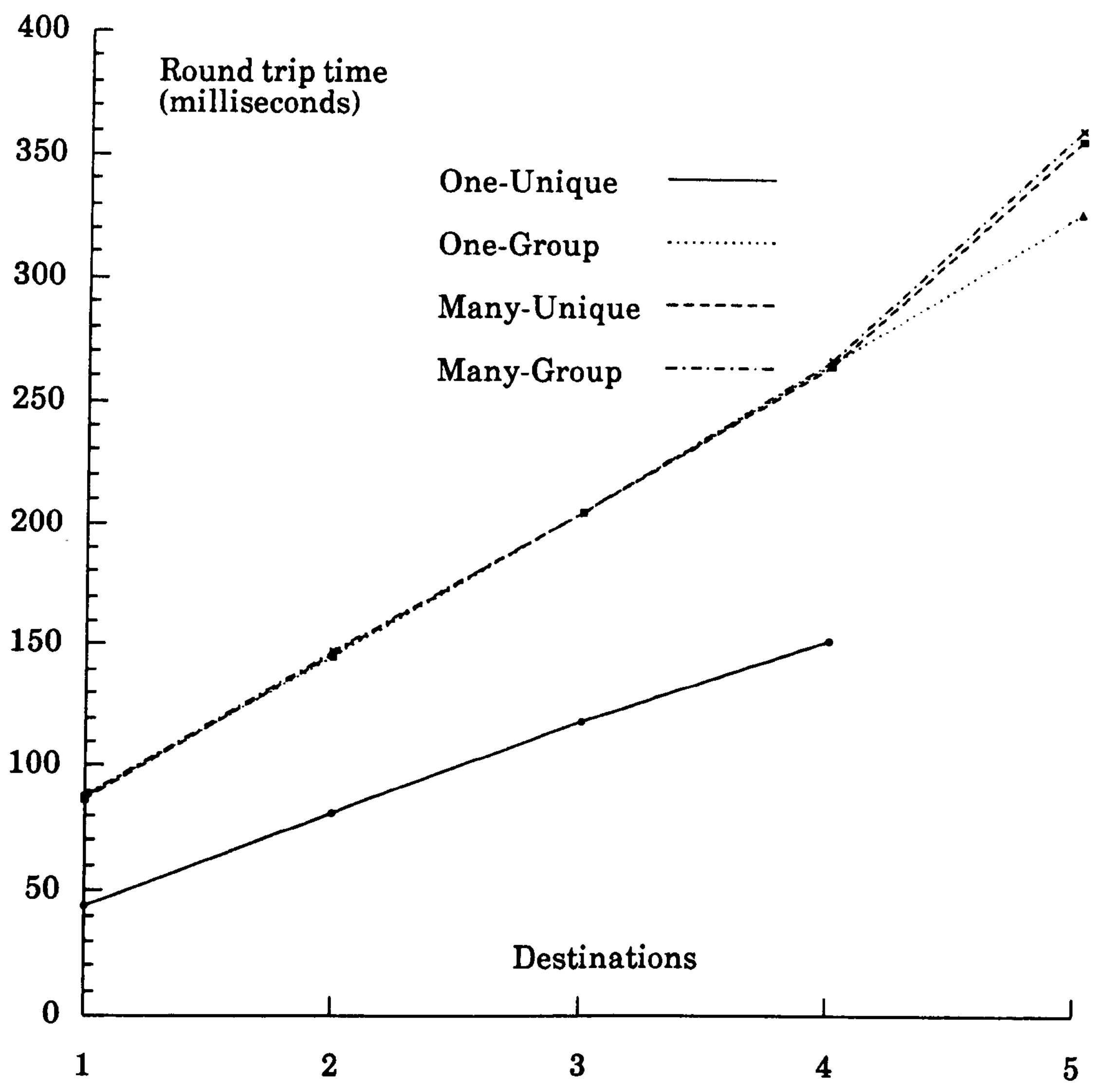

Figure 5-13: VAX-750 to Whitechapel (4 byte message)

Notes:

a) Many-Unique and Many-Group transmissions allowed at most four identifiers to be associated with a single transmission, hence the increased slope when the number of destinations is increased to five;

b) the destination Whitechapel was not able to receive more than four One-Unique transmissions - hence the number of One-Unique transmissions stopping at four.

d) in Many-Unique and Many-Group networks, additional message transmissions may be required if the number of Destinations exceed the number of identifiers that can be associated with a message (as illustrated by the increase in transmission times when increasing the number of destination Processes from 4 to 5).

These observations confirm what was suggested by the multicast transmission taxonomy – that the speed of a multicast transmission to a single Host consisting of "R" receiving Processes depends upon both the speed at which messages are distributed across the network and the speed at which messages could be distributed within the Host. In addition, the size of the message also affects the distribution time.

This point is further exemplified if transmissions from a slow Host (a Whitechapel) to a fast Host (the SUN-3) are considered. In Figure 5-14, the graph shows quite clearly that if the overall interprocess transmission time (across the network) is slow, more dramatic results can be obtained by using a multicast layer to distribute the messages.

5.4.4 Destinations on Separate Hosts

In the following set of tests, the overheads associated with distributing messages to Destinations on separate Hosts were examined. As in the tests involving destinations on a single Host, these tests entailed transmitting a multicast message to a multicast set consisting of an increasing number of receivers and determining the total trip time (i.e. Request and Response). All four intranetwork architectures were examined (One-Unique, One-Group, Many-Unique, and Many-Group).

As in the previous set of tests, the multicast set membership increased from one Destination to a maximum of five, with each new (Destination) Process being placed on a (new) separate Host. To ensure that the tests were

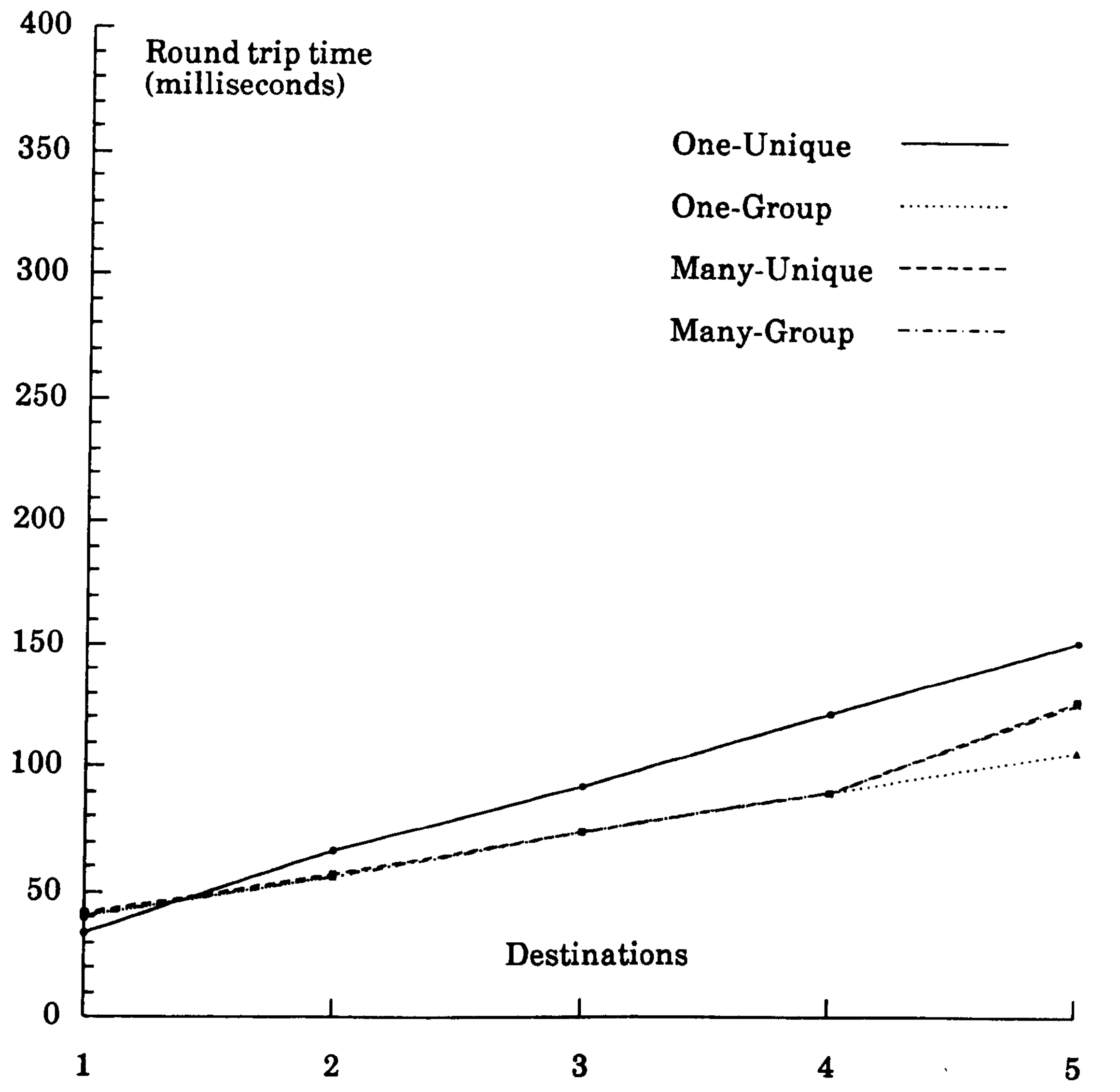

Figure 5-14: Whitechapel to SUN-3 (4 byte message)

Notes:

a) Many-Unique and Many-Group transmissions allow at most four identifiers to be associated with a single transmission, hence the increased slope when the number of destinations is increased to five.

conducted under similar conditions, the same ordering of Hosts was used for each of the different multicast transmission types being examined.

The Host ordering was from the fastest to the slowest in terms of CPU speed (that is, SUN, Orion, VAX-750, and the pair of Whitechapels). As in the previous tests, two sets of observations are presented, first, a 4-byte Request and a 4-byte Response, and second, a 512-byte Request, and a 4-byte Response. The Source Process ran on a VAX-750.

Figures 5-15 and 5-16 show the round trip times for a 4-byte message and a 512-byte message respectively. The following observations are made from these graphs:

a) the speed of the machines within the multicast set affects the overall message distribution time;

b) in a multicast transmission, the order in which the members are transmitted to is important. For example, had the tests begun with the Whitechapels, the overall transmission time would have been higher to begin with and much closer to the horizontal, since the speed of the Whitechapels would have been the limiting factor;

c) when dealing with machines of comparable speeds (for example, Hosts 2 and 3 (Orion and VAX-750) and Hosts 4 and 5 (the pair of Whitechapels)), the overheads associated with repeated One-Unique transmissions suggests that using the Multicast Communication Layer for message distribution may be attractive in homogeneous networks;

d) Many-Unique and Many-Group multicast transmission follow the same curve as does One-Group since all three use similar algorithms for message distribution. However, when the number of identifiers required to identify the members of the multicast set exceed the number of identifiers that can be associated with a single message, additional message transmissions are required, hence the (not surprising) increase in slope between Hosts 4 and 5.

All these observations confirm what was suggested by the multicast transmission taxonomy, that in general, the time taken for a message to reach

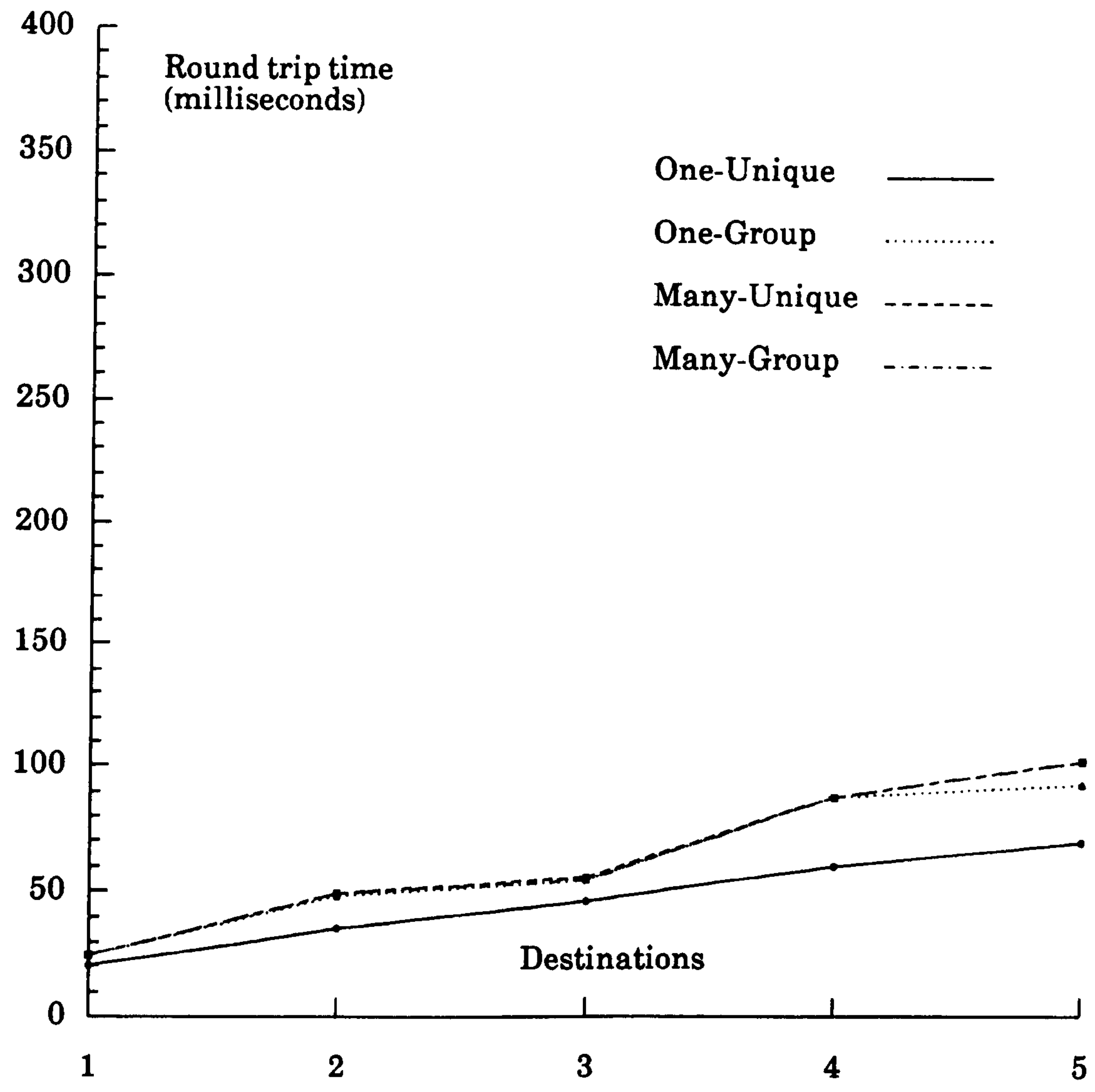

Figure 5-15: VAX-750 to Separate Hosts (4 byte message)

Notes:

a) Many-Unique and Many-Group transmissions allowed at most four identifiers to be associated with a single transmission, hence the increased slope when the number of destinations is increased to five.

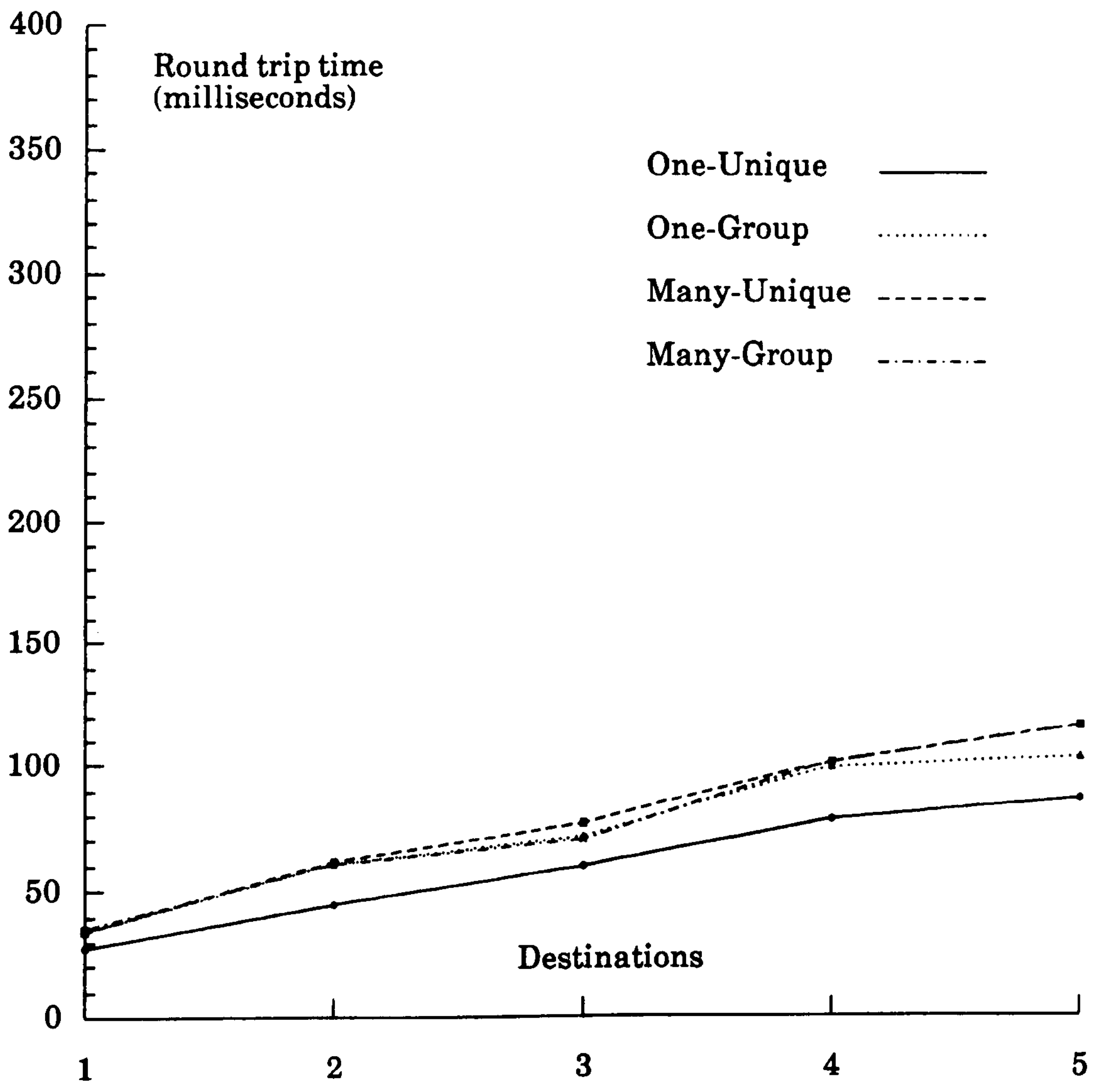

Figure 5-16: VAX-750 to Separate Hosts (512 byte message)

Notes:

a) Many-Unique and Many-Group transmissions allowed at most four identifiers to be associated with a single transmission, hence the increased slope when the number of destinations is increased to five.

all members of a multicast set is dictated by the speed of the slowest machine on the network.

5.5 Concluding Remarks

The purpose of this Chapter was to show how the proposed primitives could be implemented on a variety of intranetworks. Since the majority of the proposed intranetworks were not supported by the facilities available for use in the Computing Laboratory, it was necessary to design and implement a series of intranetwork architectures to allow the testing of One-Group, Many-Unique, and Many-Group intranetworks.

Once the intranetworks were implemented, it was then possible to implement and test the proposed primitives using the Multicast Communication Layer. The tests showed that the primitives could be implemented successfully on intranetworks supporting different numbers and types of identifier.

In addition to the comments already made in this Chapter regarding the results of the various tests, one can conclude that the time taken to distribute a message to the members of a multicast set is governed by the time required to supply the message to the slowest member of the multicast set. For example, when transmitting a multicast message to a multicast set consisting of a variety of Hosts, the speed of the slowest Host determines the overall message distribution time.

Therefore, when considering multicast communications in a layered distributed system, a network designer should consider the following:

a) avoid using intranetworks consisting of Hosts with widely varying speeds. Instead, an intranetwork supporting a homogeneous collection of Hosts should be considered, since this may result in faster distribution times. For example, if the members of the multicast set are uniformly distributed amongst the various Hosts, a single One-Group transmission would probably be faster than a series of One-Unique transmissions;

b) attempt to ensure that members of a multicast set reside on different Hosts if the cost of message distribution on the receiving Host is high. (Clearly, this is not always possible in a truly distributed system, since members of the multicast set may migrate between the various Hosts);

c) use One-Group transmissions if the members of the multicast set can be identified using a single identifier, whereas Many-Unique or Many-Group transmissions should be used if the members of the multicast set are identified with a variety of identifiers;

d) Many-Group transmissions should be considered if it appears that users will build multicast sets from other, existing multicast sets.

Chapter 6 Internetwork Multicast Communications

In light of recent trends in distributed systems, it is generally assumed that instead of using a single monolithic network, many organizations will employ a series of small networks for reasons such as security and reliability [Shepherd1985a] and because of decentralized or incremental acquisition of equipment. In certain situations these individual networks will be interconnected, thereby allowing **internetwork** communications, that is, communications *between* networks.

The point at which two (or more) networks are interconnected is known, generically, as a **Gateway**. The specific functions of a Gateway can vary, depending upon the networks it interconnects. For example, a Gateway interconnecting a pair of Ethernets may simply transmit each message it receives from one Ethernet onto the next, whereas a Gateway interconnecting a Cambridge Ring and an Ethernet may be responsible for changing message formats (from, say, an Ethernet packet into a series of Cambridge Ring mini-packets) or changing identifier formats (from 48-bit Ethernet identifiers into eight-bit Cambridge Ring identifiers) or both.

Communications can be further complicated in an internetwork multicast communication since there may now be many Destinations, rather than one, on the remote network. Consider, for example, a situation in which a multicast set consists of members on two networks. Ideally, a Source Process on one of the networks, using the multicast *send* primitive, could transmit a message to all Destinations (both local and remote):

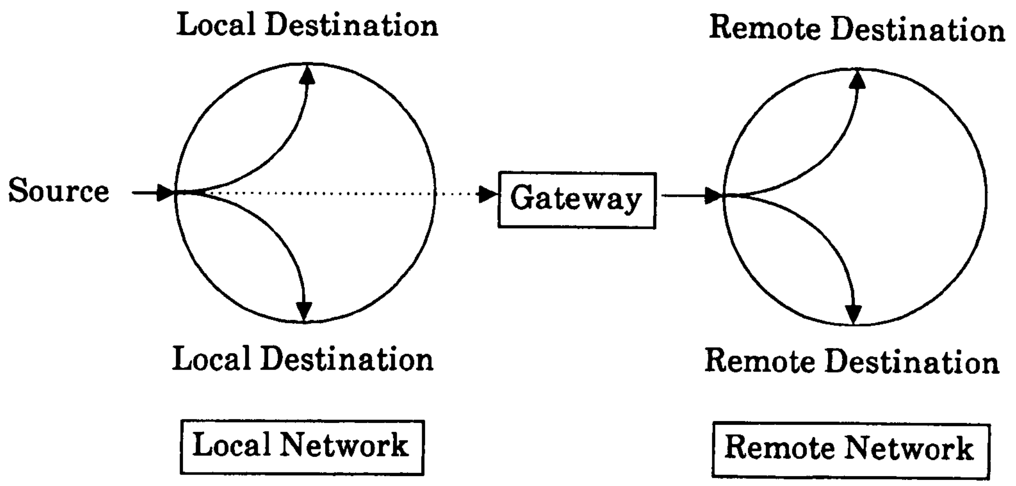

Figure 6-1: Relationship between Local Network, Remote Network and Gateway in a Multicast Transmission

However, at the Communication Layer (or the Network Layer), the task of performing the communication may not be so simple since Destination identification may differ between the two networks. For example, the Local Network may support One-Group transmissions between Hosts, whereas the Remote Network may require One-Unique transmissions. In situations such as these, a network designer is faced with problems such as:

- should the Source or the Gateway manage the list of remote Destination identifiers?
- if the list of identifiers are managed by the Source, should it transmit individual messages, via the Gateway, to the remote Destinations, or should it pass all the identifiers and a single message to the Gateway for transmission by the Gateway?
- if the Gateway is responsible for maintaining the list of remote Destination identifiers, how should the Source indicate that a certain message is to be transmitted to the members of the multicast set on the remote network?

This Chapter will attempt to answer these questions by examining:

a) how different methods of internetwork multicast communications affect the design of Gateways;

b) how the proposed primitives can be used to support internetwork multicast communications;

c) what methods of multicast transmission, if any, are best suited to internetwork multicast communications.

The Chapter is organized as follows. In the next Section, the different ways in which a Source can transmit a message to a Gateway are discussed, while in the third Section the problems associated with identifying Destinations on remote networks are examined. In the fourth Section, different methods of performing internetwork multicast communications are discussed, with an emphasis on multicast gateway design and how the proposed multicast communication primitives could be used in their design. The performances of the different types of Gateway are compared in the fifth Section. In the final Section, the findings presented in this Chapter are reviewed.

6.1 Gateway Identification

In any internetwork communication, before the message reaches the final, intended Destination, the message must pass through one or more Gateways. The exact function of the Gateway varies, depending upon the situation. For example, a Gateway connecting two identical networks (i.e. networks supporting the same protocol, identifier and message structures) need only act as a **bridge** between the networks [Shepherd1985a].

However, when connecting networks which support different protocols, identifier types or message structures, the Gateway is forced to perform additional functions such as protocol or identifier conversion. Once a Gateway has received a message, it must then attempt to transmit the message to the intended Destinations using the facilities available on the remote network. The purpose of this Section is to examine two different methods in which a Source (acting as a transmitter) can transmit a message to a Gateway.

As in the intranetwork communication case, the Source must supply the message and some form of identification which allows the message to reach the intended Gateway. There are two broad divisions of multicast Gateway identification: **implicit** and **explicit.**

6.1.1 Implicit Gateway Identification

An implicitly identified Gateway does not require a separate message transmission by the Source. Instead, it is assumed that the Gateway is able to determine the Destination(s) from the identifier used to reach the Gateway. For example, if the Source transmitter performed a One-Group multicast transmission on the local network, an implicitly identified Gateway would be expected to receive a copy of the message. From the single Group identifier used to identify the members of the multicast set on the Source's (local) network, the Gateway would then be expected to identify all the members of the multicast set on the remote network.

In the V-System [Cheriton1983a], the Gateway is implicitly identified since multicast messages are broadcast to all Hosts (including Gateway Hosts). Upon receipt of a message, a V-System Gateway determines if the message should be transmitted onto the Destination network by inspecting the Group identifier. If the Gateway determines that the multicast set exists on the Destination network, the Group identifier (supplied by the Source) is mapped into an equivalent Group identifier (used by the members of the remote multicast set) and transmitted on the Destination V-System network.

6.1.2 Explicit Gateway Identification

When the Gateway is to be identified explicitly, the Source must transmit a separate message (explicitly) to the Gateway in question (in addition to those sent to the local Destinations). For example, on a network supporting One-Group multicast transmissions, at least two transmissions would be required, one to the local Destinations making up the local multicast set, accessible using a One-Group transmission, and the other to the Gateway, using a One-Unique transmission. (Note, this assumes transmission to a single Gateway, if there were more Gateways, additional One-Unique transmissions might be required or possibly a single One-Group transmission to Gateways belonging to a Gateway group.) When a Gateway is explicitly accessed, it is assumed that the Gateway requires additional routing information to enable it to transmit the message to the intended Destinations:

```
GatewayMessage =
record
    GwId : IdStructure;
    FDId : IdStructure;
    Data : array [0 .. MaxSize] of Bytes;
end;
```

where:

GwId: the Gateway Identifier, which allows the routing of the message to the Gateway;

FDId: the Final Destination Identifier, which allows the routing of the message, by the Gateway, to the intended Destinations. Depending upon how the Gateway identifies the intended Destinations, the identifier combination can be any of One-Unique, One-Group, Many-Unique, or Many-Group (how these identifiers can be used will be discussed in the next Section);

Data: the information to be transmitted.

For example, in MP [Ahamad1985a], Gateways are explicitly identified by the Source. The Source performs two transmissions, the first on its local network (an Ethernet, using a One-Group transmission) and the second to the Gateway in question. The Gateway is supplied with the message and a single Group identifier which the Gateway maps into another Group identifier for transmission on the remote network (another Ethernet).

6.1.3 Summary

In this Section, two methods of Gateway identification have been discussed: implicit identification and explicit identification. From this discussion we make the following observations:

a) implicit identification of the Gateway minimizes the number of transmissions required on the Source network (since, ideally, only one transmission is required). However, the Source has no method of controlling which of the Destinations receive a copy. For example, a Source could not transmit a message to only those Destinations on its own local network since the message is transmitted by all Gateways to all Destinations. This problem is further illustrated in a situation where a Destination can be reached by two implicitly identified Gateways - the Destination in this example can receive two copies of the same message.

b) explicit identification of the Gateway avoids the above problem by requiring the Source to explicitly identify the Gateways through which the transmission is to occur. This obviously results in additional message generation on the Source network, a potentially undesirable feature, if, for example, there are multiple Gateways and each must be explicitly identified.

In the next Section, an examination of how the different types of Gateway identification affect the identification of the members of the multicast set on the remote network is presented.

6.2 Identifying the Destination Members of the Multicast Set

In addition to identifying the Gateway, it is also necessary for the transmitting Source to identify the intended Destinations, both local and remote. Obviously, the initial identification of the particular multicast set in question is determined by the Source using either Unique or Group identifiers. However, the maintenance of the identifier(s) used to indicate the members of the multicast set can be performed by either the Source or the Gateway. For example, if the members of a multicast set on a certain remote network are identified using Unique identifiers, the management of the list of these identifiers may be the responsibility of either the Gateway or the Source.

In this Section, some of the problems relating to Source and Gateway multicast set membership identification are examined.

6.2.1 Multicast Set Identification by Source

Source membership identification means that the Source maintains the actual identifiers of the Destinations (either Unique or Group) rather than the Gateway. For example, if the members of a multicast set are to be accessed by a series of One-Unique transmissions, the Source would maintain the list of Unique Destination identifiers. This does not necessarily mean that the Gateway is passive and simply forwards each message it receives onto the Destination network as the following examples illustrate:

a) the Gateway can act as a bridge, simply transmitting the supplied message on the remote network. For example, in the worst case, if the multicast set members on the remote network require One-Unique transmissions, the Source would be required to transmit the same message to each Destination on the remote network through the Gateway. Similarly, if the Destination network allowed One-Group transmissions, the Source would only be required to transmit a single message via the Gateway to the remote Destinations.

b) situations such as those described above, in which the Source must transmit individual messages to the intended Destinations, can be avoided if the Gateway is given additional functionality. For example, if the intended (remote) Destinations can only be accessed using One-Unique transmissions, the number of Source transmissions can be reduced if the Gateway is supplied with the message and a list of many unique identifiers. The Gateway can then be responsible for transmitting the message to the individual Destinations.

c) if the Gateway is promiscuous (i.e. it receives all messages transmitted on the Source network – implying implicit Gateway identification), it can also act as a bridge, forwarding messages to the remote network. Given additional functionality, the Gateway could filter those messages not intended for the Destination network (for example, messages for multicast sets not supported on the remote network could be discarded).

6.2.2 Multicast Set Identification by Gateway

Although Gateway multicast set identification implies that the actual multicast set identifiers are maintained by the Gateway, the Source is still (not surprisingly) responsible for supplying the message and indicating which multicast set is to be transmitted to. For example, the members of a multicast set on a remote network may require a series of One-Unique transmissions, but the Source may use a single Group identifier to indicate the identity of the multicast set to the Gateway. It would therefore be the responsibility of the Gateway to map the supplied Group identifier into the Unique (remote) Destination identifiers.

When the Gateway is used to identify the intended Destinations of the multicast set, the Source can use either implicit or explicit Gateway identification, since both techniques supply the Gateway with the identifiers which allow the transmission to occur. Both the V-System and MP are examples of multicast set identification by the Gateway.

6.2.3 Comparing Source and Gateway Destination Identification

If the objective of the multicast implementation is to minimize the amount of internetwork traffic on the Source network (i.e. the traffic between the Source and the Gateway), an internetwork transmission technique should be used which produces the minimum number of Source messages. For example, by using a promiscuous Gateway (which receives copies of *all* messages) and identifying the remote Destinations with a single group identifier, the minimum number of messages will be sent across the local network to the Gateway (ideally, one). This technique is used by the V-System.

However, the above technique does have certain drawbacks. For example, the technique does not allow the Source to perform a multicast transmission on its local network only, avoiding Gateway transmissions, since the Gateways always transmit the multicast messages they receive onto remote networks. The obvious solution to this problem is to use a separate multicast identifier for messages on the local network and another for messages destined to the Gateway. This technique is used by MP.

There are other arguments for having the Source maintain a list of Destination identifiers. For example, by maintaining a list of the Unique identifiers of all possible members of a multicast set, the Source is able to determine which Destinations are responding. For example, if responses are expected from all Destinations, the Source must maintain a list of the Unique identifiers of each of the possible Destinations. However, it would seems more sensible to develop protocols which, as a first step, build a list of Destination identifiers based upon the Destinations which respond to the initial transmission, rather than attempting to rely on possibly outdated multicast set membership information. By taking this approach, it becomes apparent that it may not be necessary for the Source to maintain anything other than a single Group identifier which is used by the Gateway to identify the intended remote Destinations, mapping the Group identifier into the required number and type of identifiers of the remote network. Not only does this approach result in the minimum of traffic on the local network, it also isolates any network inefficiencies on the remote network. For example, if the remote network only supports One-Unique transmissions, having the Gateway perform the mapping (such as One-Group to One-Unique) would keep the One-Unique transmission(s) on the remote network.

6.2.4 Summary

From the discussion in this Section, one can make the following observations:

a) identifying internetwork multicast transmissions separately from the intranetwork multicast transmissions allows additional flexibility in that the Source can selectively perform its multicast transmission;

b) Source network traffic can be kept to a minimum if remote Destination multicast set members are identified using a Group identifier;

c) network transmission inefficiencies should not be allowed to migrate beyond the network on which they exist.

The next Section contains an examination of how Gateways such as those proposed in this Section could be implemented using the multicast communication primitives and what changes, if any, would be required of the primitives.

6.3 The Design, Implementation and Testing of Several Multicast Gateways

In this Section, the design, implementation and testing of several multicast Gateways using the multicast communication primitives developed in

Chapter Four is presented. In addition, extensions to the existing primitives to allow the implementation of the Gateways will be discussed when required.

6.3.1 Implicitly Identified Gateways

6.3.1.1 Design Considerations

Implicitly identified Gateways are to receive all (multicast) messages transmitted on a specific network. This is equivalent to joining *all* possible multicast sets which may exist on a network and receiving all multicast messages transmitted on the network. An implicitly identified Gateway could therefore be written as follows (using the existing multicast communication primitives):

```
join (ALL);
repeat
    receive (ALL, Message, NULL, Indefinite);
    OutgoingIds := map (IncomingIds);
    if OutgoingIds < > NULL then
        send (OutgoingIds, Message, sizeof(Message));
until
FailureDetected;
leave (ALL);
```

In the above example, the Gateway first joins all possible multicast sets (with a *join* (ALL) - see Section 6.3.1.2 on Primitive Changes). Any multicast message received from the Source's network is made available to the Gateway. The incoming multicast set identifier is then mapped into the equivalent multicast set identifier used on the remote network, if it exists. The message is then transmitted on the remote network. This cycle continues until an error is detected.

The *map* function is intended to take the incoming identifier(s) and, from their values, produce the equivalent identifier(s) used on the remote network. The operations performed by the *map* function depend upon the identifier(s) available to the Gateway from the Source and how the remote Destinations are identified.

For example, in the simplest case, if the remote Destinations are identified by the same Group identifier as the local Destinations (and the Gateway), the *map* function would not be required to make any changes to the identifier. However, if the Gateway can only be uniquely identified (i.e. the *join* (ALL) primitive supplies a name server with the Unique identifier of the Gateway), a protocol must be devised to ensure that the Gateway not only receives the message, but also receives an identifier(s) to allow identification of the remote Destinations.

Clearly, in situations other than where the local and remote multicast set identifiers are identical, the *map* function must have access to the multicast set identifiers of the remote network. This could be achieved by using the *getid* primitive. For example, if the incoming identifier could be used to represent a name, the name could be supplied to the *getid* primitive, which could, in turn, supply the identifier(s) associated with the intended Destinations.

To avoid the cycle of accessing the name server (or the file containing the Destination identifiers) each time a message for the same multicast set is received, the *map* function should also have the ability to cache the most recently used identifiers. This issue will be discussed further in Section 6.3.1.3.

6.3.1.2 Primitive Changes

The implicitly identified Gateway presented in Section 6.3.1.1 illustrates some of the limitations of the existing primitives with respect to internetwork multicast communications.

For the implicitly identified Gateway to function, it must join *all* possible multicast sets. If the present implementation of the *join* primitive were to be used by the Gateway, the Gateway would be required to join each multicast set individually – a potentially slow and error prone activity given that the number of multicast sets could be changing over time. However, by modifying the *join* primitive it is possible to allow a Process to join *all* multicast sets:

a) a special multicast set identifier, *ALL*, is required which indicates to the *join* primitive that all multicast sets are to be joined;

b) how the primitive is implemented clearly depends upon how the Gateway is identified. For example, if the Gateway is uniquely identified, a name server must be informed of the Gateway's (Unique) identifier and that the identifier is to be supplied with every request for a multicast set that is made. However, if the Multicast Communication Layer can receive all Network Layer messages (for example, all multicast messages are broadcast to all Hosts), then the Multicast Communication Layer must be informed (using the *join* primitive) to supply *all* received messages to the Gateway.

Upon receipt of a message (either through a filter or directly from the *receive* primitive), the Gateway has no indication of the multicast set to which the message was intended since the *receive* primitive supplies only the message, its size and the identifier of the Source. Therefore, the *receive* primitive must be modified in order to allow it supply the multicast set identifier.

Multicast set identifiers do not indicate the network to which they refer. That is, when a Gateway transmits or receives using a multicast set identifier, the Multicast Communication Layer has no indication as to the network the identifier is intended for. Therefore, multicast set identifiers must be modified to permit the *send* and *receive* primitives to determine the network in question.

Finally, the *send* primitive must be modified so that the (Source) identifier supplied with the message is the Source identifier of the original transmitter, not the Gateway's identifier. This is to ensure that the Destination receives the identifier of the original transmitter of the message, not the identifier of an intermediate Gateway.

6.3.1.3 Implementation and Testing

In order to examine implicitly identified Gateways and test the modified primitives described in this Section, an internetwork environment was required. However, the facilities available in the Computing Laboratory made such tests impossible to perform directly since only a single Ethernet existed (without any bridges or Gateways).

Fortunately, it was possible to create logically separate networks by assigning different Port numbers to different Multicast Communication Layers, thereby ensuring that multicast transmissions on one "network" were not received on the other "network". For example, for intranetwork testing, One-Group transmissions were sent to the Multicast Communication Layers associated with Port 9999. By using a different Port identifier for a One-Group Multicast Communication Layer on the second "network", say 8999, it was possible to partition the network:

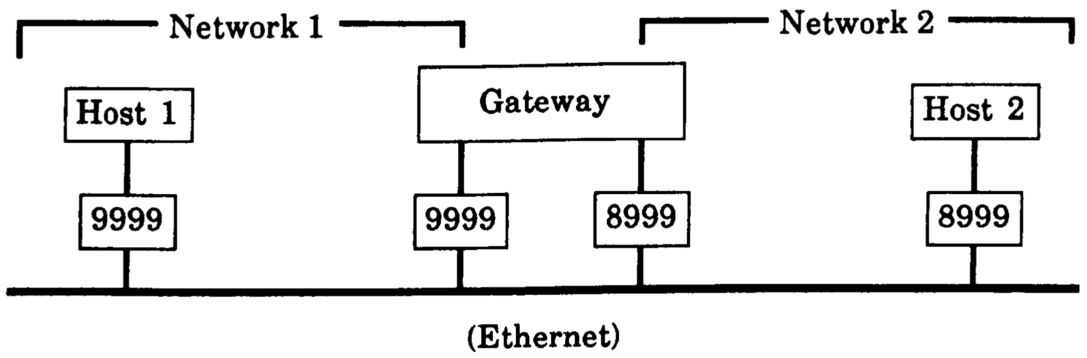

Figure 6-2: Logical Partitioning of Ethernet into Two Networks

The above architecture was used to examine two different implicitly identified Gateway configurations:

- a One-Group transmission network to a One-Group transmission network, and
- a One-Group transmission network to a One-Unique transmission network.

The first, One-Group to One-Group, was implemented as a simple bridge requiring the minimum amount of processing on the part of the Gateway. Briefly, an incoming local network multicast set identifier available to the Gateway using the modified *receive* primitive was mapped into the equivalent multicast set identifier of the remote network by the *map* function. This new identifier was then used for transmitting the message on the remote network. Note that the *send* primitive was also modified to allow the Gateway to indicate that the Source identifier was not to be altered when transmitted.

The second implementation examined a Gateway in which the supplied multicast set identifier was mapped into a series of Unique identifiers. In this example, the incoming multicast set identifier from the Source was first turned into an ASCII string and used by the *getid* primitive to obtain the list of unique identifiers associated with the remote Destination members of the multicast set. Once the list of Unique identifiers were made available, the Gateway performed a multicast transmission to the multicast set consisting of the uniquely identified remote Destinations.

A simple identifier caching method was also developed whereby the list of identifiers associated with the remote multicast set would be kept in the Gateway's main memory, as opposed to a file or a name server, for about thirty minutes. If no messages were received for the multicast set after the expiration of the time period, the storage was released. For example, in the second Gateway examined (One-Group to One-Unique), upon receipt of a multicast set identifier, the *map* function would examine a list multicast set identifiers. If the incoming multicast set identifier was not found, the *getid* primitive would be used to generate the list of unique Destination identifiers. The list of unique identifiers were then stored in the list of multicast set identifiers:

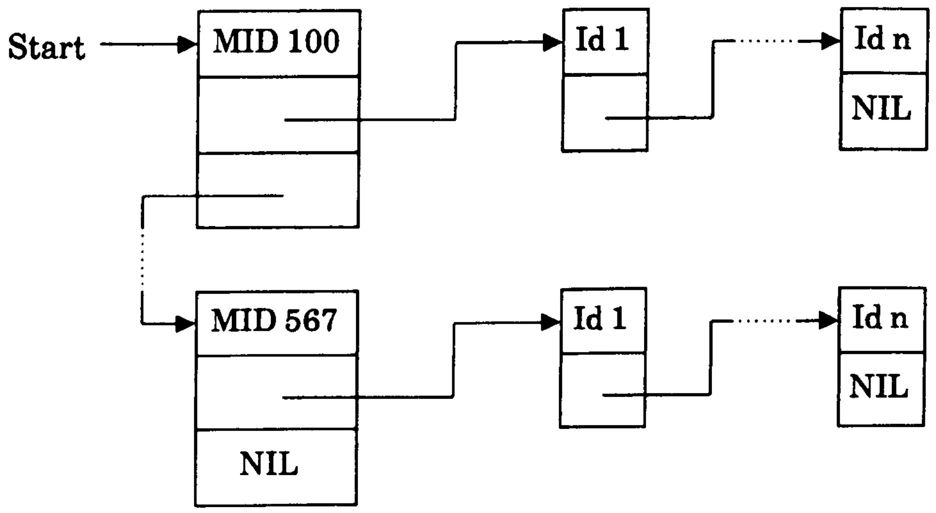

Figure 6-3: Cached Identifier Lists

When subsequent (identical) multicast set identifiers were received, the list would be scanned again, and the list of unique Destination identifiers would be made available (without using the *getid* primitive).

Although the cached identifier list does allow the Gateway faster identifier accessing, it can lead to difficulties. For example, if the membership of the multicast set changes over a period of time, the cached identifier list could become out of date. This problem is overcome by shortening the time cached lists were kept in main memory.

6.3.2 Explicitly Identified Gateways

6.3.2.1 Design Considerations

Explicitly identified Gateways only receive those messages which are explicitly transmitted to them. A network designer can therefore implement a Gateway so that it can either join a gateway multicast set (and wait for messages sent to the multicast set) or wait for messages sent directly to the Gateway (i.e. a unicast transmission). In the following example, the explicitly identified Gateway joins a specific multicast set:

```
join (GatewayGroup);
repeat
    receive (GatewayGroup, Message, NULL, Indefinite);
    OutgoingId := map(SuppliedIds);
    if OutgoingId < > NULL then
        send (OutgoingId, Message, sizeof(Message));
until
FailureDetected;
leave (GatewayGroup);
```

(Note, the above Gateway could have been implemented as a uniquely identified Gateway – requiring the Source to send individual messages to each Gateway.)

In the above example, once the Gateway has joined the GatewayGroup multicast set, it waits for messages transmitted with the Gateway multicast set identifier. Upon receipt of a message, the identifier(s) supplied with the message are then mapped into the remote Destination identifiers. If the (remote) identifiers exist for the multicast set indicated, the Gateway transmits the message on the Destination network. This cycle is repeated until a failure is detected.

As with the implicitly identified Gateways, the *map* function is to generate the remote Destination identifier(s) from the supplied identifiers.

6.3.2.2 Primitive Changes

Other than the requirement that the multicast set identifier indicate the network to which it refers and the *send* primitive not overwrite the Source identifier, no changes were required of the multicast primitives. In addition, the "ALL" identifier was no longer required.

6.3.2.3 Implementation and Testing

The test internetwork environment described in Section 6.3.1.3, whereby the Ethernet was logically partitioned into subnetworks by using different Port numbers for the same multicast layer on the different "networks", was used in the implementation and testing of explicitly identified Gateways.

Three different types of internetwork communications were considered:

- the Source supplies a Group identifier, while the Destination network supports One-Group transmissions;
- the Source supplies a Group identifier, while the Destination network supports One-Unique transmissions;
- the Source supplies a series of Unique identifier(s), while the Destination network supports One-Unique transmissions.

In the first implementation, Group identifier to One-Group internetwork communications, the supplied Group identifier was mapped into the Group identifier of the intended remote Destinations, in the same manner as described for One-Group to One-Group in Section 6.3.1.3. However, unlike the implicitly identified Gateway which had to be identified by the multicast set identifier of the local network, the explicitly identified Gateway permitted the

Source to supply actual identifier of the remote multicast set – thereby eliminating the overheads associated with the *map* function.

The second implementation, Group identifier to One-Unique internetwork communications, was implemented in the same fashion as that described for the One-Group to One-Unique implementation in Section 6.3.1.3. The technique of caching the identifiers was also used in this implementation.

In the final implementation, the Source supplied the actual identifiers of the intended remote Destinations. Two implementations were examined. The first involved the Source sending each message with a single Unique Destination identifier which was supplied directly to the *send* primitive by the Gateway with the message for transmission on the remote network, while in the second implementation, the Source supplied a series of Unique identifiers with the message, thereby requiring the Gateway to perform multiple transmissions. In both of the implementations examined, the *map* function was not required, since the Source maintained the list of the actual Destination identifiers.

6.3.3 A Hybrid Implementation

A hybrid Gateway, combining some of the features of both implicitly and explicitly identified Gateways was also considered in the examination of internetwork multicast communications as an attempt at reducing the overheads associated with mapping the incoming identifier(s) into the equivalent remote network identifiers.

In the hybrid Gateway, two Gateways were required. The first, the *parent* Gateway, was designed to accept an explicitly identified message from a Source containing the multicast set identifier used by the Source to identify the members on the local network. This identifier was mapped into the identifier(s) used by the members of the remote multicast set. At this point, a *child* Gateway Process was spawned which joined the multicast set on the local network using the *join* primitive and the multicast set identifier supplied by the Source:

```
getid ("GatewayGroup", ParentGateway);
join (ParentGateway);
repeat
    receive (ParentGateway, Message, SetUpMessageOnly, Indefinite);
    RemoteIds := map (Message ^. LocalMID);
    spawn (Child, Message ^. LocalMID, RemoteIds);
until
NetworkError;
leave (ParentGateway);
```

Thereafter, the Source transmitted all messages with the local multicast set identifier, a copy of which would be received by the child Gateway, acting as an implicitly identified Gateway. Since the child Gateway already has the remote multicast set identifier (supplied by the original parent Gateway), the overheads associated with the *map* function were eliminated – resulting in a faster Gateway:

```
join (LocalGroup);
repeat
    receive (LocalGroup, Message, NoFilter, CacheLimit);
    send (RemoteIds, Message ^. Data, Message ^. Size);
until
NoMoreMessages;
leave (LocalGroup);
```

To ensure that the child Gateways did not exist indefinitely, they were "cached" in much the same way as the identifier list (described in Section 6.3.1.3). That is, the *receive* primitive was timed – if a message was not received from the Source within a thirty minute period, it was assumed that the Source was no longer transmitting messages to the remote multicast set and the child Gateway was terminated. To give the Source the opportunity to reestablish contact with the members of the remote multicast set, via the parent Gateway, the child Gateway was designed to inform the Source of its termination.

Subsequent versions of the hybrid Gateway have been developed in which the Source transmits a special Gateway message indicating that the child Gateway should be terminated by the parent Gateway (in much the same way a virtual circuit is closed once it has been determined that the communication is to stop). As an example, consider the following Source Process:

```
getid ("GatewayGroup", Gateway);
getid ("MulticastGroup", Destinations);

Message . Type := SetUpGateway;
Message . ID := Destinations ^. MID;
send (Gateway, Message, sizeof(Message));

{ Child Gateway setup - transmit to the multicast set "Destinations" }
repeat
    ...
    send (Destinations, Data, sizeof(Data));
    ...
until
AllDone;

Message . Type := ShutDownGateway;
Message . ID := Destinations ^. MID;
send (Gateway, Message, sizeof(Message));
```

The Source initially transmits a SetUpGateway message to the Gateway(s) belonging to the "Gateway" multicast set. Thereafter, all messages sent to the multicast set "Destination" are received by both the members of the local multicast set and any child Gateway(s). Once the Source has finished transmitting to the multicast set, it indicates that no more messages will be sent with a ShutDownGateway message.

The hybrid Gateway has several advantages over the other types of Gateway discussed in this Chapter:

a) no changes are required to the *join* primitive;

b) the "parent" Gateway only receives requests to spawn "child" Gateways - thereby reducing its workload (unlike the implicitly identified Gateway which receives all multicast messages, including those without members on the remote network);

c) the Source Process has the option of including the remote networks in its multicast transmission since the Gateway must be explicitly identified to set up the child Gateway;

d) the overheads associated with constantly mapping the incoming identifier into a remote identifier are eliminated, since the child Gateway is only associated with a single multicast set.

Finally, for those transmissions which, because of size, do not warrant the setting up a child Gateway, a Source should have the ability to bypass the spawning Gateway either by sending its messages to a different Gateway or indicating to the spawning Gateway that the message should be transmitted directly onto the remote network with the appropriate mapping.

6.3.4 Summary

In this Section several different multicast Gateway designs have been considered, all based upon either implicit or explicit Gateway identification. Both the implicitly and explicitly identified Gateways had certain overheads associated with them, making many of them costly in terms of the amount of processing required by the Source (e.g. transmitting a series of One-Unique messages from the Source), or the amount of processing performed by the Gateway (e.g. repeated scanning through lists of multicast set identifiers). Similarly, the amount of traffic produced on the local network could affect the overall performance of the communication (e.g. in a congested network, explicitly identifying the Gateway could prove an expensive proposition since additional messages are required from the Source to the Gateway).

It was found that many of these problems were eliminated by using a hybrid Gateway which allowed an explicitly identified "parent" Gateway to spawn a "child" Gateway which acted as an optimized implicitly identified Gateway. The price for this feature was an addition protocol on the part of both Source and Gateway.

6.4. Performance Results

In order to allow a comparison of the three different types of Gateway described in this Chapter, a simple round trip test (similar to that described in Chapter Five for comparing the different intranetwork configurations) was devised. This test consisted of a Source Process transmitting a message to a Gateway, which then transmitted the message to the destination multicast set on the "remote" network using a One-Group multicast transmission. The Destination Process then responded directly to the Source Process:

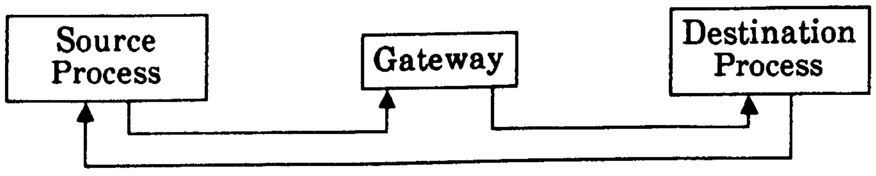

Figure 6-4: Gateway Testing Configuration

Each Gateway was subjected to the same test, notably the transmission of one thousand 512-byte messages. In all tests, the Gateway's map function performed a simple caching operation, mapping the supplied identifier into the Destination's multicast set identifier from a list containing a single identifier The average time taken for the round trip of each message is shown in Figure 6-5. Two sets of tests were performed, the first was to a "slow"

Gateway (a Whitechapel) while the second was to a "fast" Gateway (the SUN-3). In all tests, the Source Process resided on a VAX-750 and the Destination Process was on the Orion.

Gateway Host	Type of Gateway			
	Implicit (Round trip)	Explicit (Round trip)	Hybrid	
			Round trip	Setup Time
Whitechapel	142	161	142	1370
SUN-3	78	97	77	790

Figure 6-5: Comparison of Gateway Speeds

The following observations can be made regarding the results from Figure 6-5:

- as one would expect, the faster the Gateway, the faster the throughput;
- Implicit Gateways are, not surprisingly, the fastest, since only one transmission is required by the Source;
- Explicit Gateways are the slowest, since two transmissions are required in order that both local and remote Destinations receive a copy of the message;
- Hybrid Gateways are as fast, if not faster, than Implicit Gateways – if the time required to set the Child Gateway is ignored. This is because the child Gateway no longer requires the *map* function.

From these observations, one can conclude that if the number of messages to be transmitted is small, the overheads associated with using the Hybrid Gateway may make using the Implicit Gateway more attractive to use. However, if the Gateway is to cache a large number of identifiers, the time required to search for each identifier within this list may make the Hybrid Gateway faster than the Implicit Gateway.

6.5 Concluding Remarks

The objective of this Chapter was to consider different methods of internetwork multicast communications and to determine which, if any, offered the best form of (multicast) communications.

Two basic techniques were considered – those in which the Gateway was explicitly identified (i.e. the Source explicitly transmitted a message to the Gateway, in addition to those already transmitted on its local network) and implicit identification (i.e. the Gateway was promiscuous and received all multicast messages transmitted on the local network).

From this examination, the following observations were made:

a) irrespective of how the Gateway is identified, any network inefficiencies on the remote Destination network should be isolated to that network. For example, a Source on a One-Group network should not be expected to repeatedly transmit a message to Destinations on a One-Unique network since this can lead to network congestion on the local (Source) network;

b) if a Source expects to transmit a large number of messages through a Gateway, the proposed hybrid multicast Gateway should be considered since it reduces the amount of work required by the Gateway (and potentially the Source);

c) protocols should be developed which permit a Source to use a group identifier to identify the intended Destinations, but use a list of unique identifiers for Destination identification.

d) the proposed multicast communication primitives could, with minor changes, be used with multicast Gateways;

e) although implicitly identified Gateways resulted in less traffic on the local (Source's) network, all multicast messages sent on the local network were also distributed on remote networks by the Gateways. This problem was overcome using explicitly identified Gateways, at the expense of having additional traffic on the local network.

After several detailed examinations of different Gateway implementations, a hybrid Gateway was developed. The hybrid Gateway was attractive in that it allowed the Source to determine whether the message should be sent to remote networks (an explicitly identified Gateway feature), but the actual message transmission was performed by implicitly identifying the Gateway. The cost of this simple extension was the development of a protocol which required the Source to inform the Gateway that messages were to be supplied for a specific multicast set on a remote network.

Chapter 7
Concluding Remarks

The purpose of this Chapter is threefold. First, to review the issues and concepts relating to multicast communications in light of the original aims of the thesis. Second, to discuss the research presently underway at the Computing Laboratory which is utilizing the work described in this thesis. And finally, to suggest directions for future research into multicast communications.

7.1 Discussion

The following Section reviews the aims originally outlined in the Introduction to the thesis and presents a discussion of these aims in light of the various issues regarding multicast communications that have been examined.

7.1.1 Multicast Taxonomies and Classification Schemes

The first aim was to devise taxonomies which would allow network designers to describe different types of multicast communications. Three taxonomies were developed.

The first of these taxonomies, a multicast transmission taxonomy, was developed to permit the description of multicast transmission in terms of the types of identifier and the numbers of identifier that can be associated with a message. Initially, the multicast transmission taxonomy was used for describing the costs of performing multicast transmissions in a single layer of a distributed system (using the Transmitter-Receiver model). However, by applying the taxonomy to a layered architecture, it was possible to describe the actions required in many different situations requiring multicast communications, such as inter-layer communications, communications with Gateways, and message distribution.

The second taxonomy described different methods of multicast response handling using a Source-Destination model. This taxonomy was simply an enumeration of the different types of Response that a Source could expect to receive after transmitting a Request. When applied with the types of identifier possible in a multicast transmission, the taxonomy proved a useful tool for comparing how different distributed systems and networks allow a Source to determine which members of a multicast set received a copy of a previously transmitted message.

The final taxonomy used a Host-Port model and the multicast transmission taxonomy to describe multicast communications in distributed systems where transmitting and receiving entities were identified using a pair of identifiers (one to identify the intended destination Host, the other the Port, to which the entity is associated). This taxonomy proved particularly useful in the development of the various multicast communication layers using UNIX sockets.

All of the taxonomies developed in this thesis have been shown to be useful in the analysis and comparison of various existing and proposed multicast schemes, as well as suggesting other interesting possibilities such as the development of the multicast communication primitives.

7.1.2 Multicast Primitives

The second aim of this thesis was to develop a set of multicast primitives that would be implementable on a variety of distributed systems and networks, without being specific to any particular distributed system or network. By using the multicast transmission and reception handling taxonomies as well as considering several different multicast communication applications, a set of eight multicast communication primitives were developed.

The primitives were described in terms of multicast communications and multicast set management. The primitives for multicast communication (*send* and *receive*), unlike "unicast" primitives, supported both unicast and multicast communications directly. The *receive* primitive, unlike the receive primitives in other multicast implementations, allowed a "filter", enabling the Receiver to indicate which messages, if any, were to be accepted (the development of the *receive* primitive was a direct result of the multicast reception taxonomy).

The multicast set management primitives were developed to allow an entity (such as a Transmitter or a Receiver) to have control over the membership of multicast sets. Although many of the multicast set management primitives developed were similar to primitives developed elsewhere, the basic design did allow implementation on a variety of networks. For example, the *newid* primitive was developed in such a manner as to allow the creation of new, unique multicast set identifiers without requiring additional network traffic to determine if it was unique.

Finally, although the intention of the development of the primitives was to make them independent of any particular network, it was shown that in some situations, it was necessary to have access to the type of identifier used to identify the member of a multicast set. The *bestid* primitive was an example of such a case, returning either a multicast set identifier or a unique identifier, depending upon the type of network being used.

7.1.3 Implementations and Results

The final aim of the thesis was to demonstrate that the proposed multicast primitives could be supported on a variety of distributed systems, intranetworks, and internetworks.

Before any implementation of the primitives could take place, it was necessary to develop a set of four intranetworks. This was done by adding a layer of software onto the existing UNIX socket software to permit the emulation of the intranetworks. With the design of the multicast communication layer complete, it was then possible to implement and test the multicast primitives on a variety of intranetworks.

The intranetwork tests confirmed many of the ideas suggested by the multicast transmission taxonomy, such as the speed of a multicast transmission is governed by the speed of the network and the speed at which messages can be distributed on the receiving Host. In addition, from the observations made of the internetwork tests, a set of guidelines were proposed, permitting a network designer to tailor the type of multicast implementation to the equipment available.

With minor modifications, it was shown that the primitives could be extended to support internetwork multicast communications. It was shown that in an internetwork multicast communication, two types of Gateway identification were possible - implicit, where the message sent by the Source was received by the Gateway without a special transmissions and explicit, in which the Source sent messages to both the members of the multicast set and the Gateway. Although both methods were shown to have advantages and disadvantages, such as the number of transmissions required and the changes required to the primitives, both required a mapping function that added additional overhead to the Gateway, since each incoming local identifier (usually) had to be mapped into one or more remote identifiers.

An interesting outcome of the examination of internetworks was the development of a "hybrid" multicast Gateway which combined many of the features of both the implicitly and explicitly identified Gateways. With minor changes to the primitives and the development of a multicast Gateway protocol, efficient internetwork multicast communications were shown to be possible.

7.2 Ongoing Research

One of the benefits of the research described in this thesis has been the development and implementation of a multicast communication facility which has proved suitable for the ongoing research into reliable distributed systems at the Computing Laboratory. A version of the Multicast Communication Layer has been implemented and optimized to support One-Group intranetwork multicast communications using the Computing Laboratory's Ethernet. The major optimization of this Multicast Communication Layer is the restriction that the members of a multicast set are all identified with a single (One-Group) identifier. At present, the principal user of the new Multicast Communication Layer (commonly known as MCL) is a research project examining remote procedure calls and orphan detection to multiple destinations[Shrivastava1986a].

In addition to specific research projects using the multicast communication facilities, the author has developed several applications intended specifically as "communication tools" for multicast communications. These applications include [Hughes1986a]:

a) a conference call facility which allows any number of users to simultaneously communicate with one another;

b) a file distribution service which transmits a copy of a file from one machine to the set of machines specified by the user;

c) a facility to list the names of the currently active users are on the different machines on the network.

These applications have been developed to demonstrate other potential uses of multicast communications.

7.3 Directions for Future Research

As well as the ongoing research and applications described in the previous Section, there are other areas of communications which can both benefit from and contribute to the study of multicast communications. For example, another area of research soon to be undertaken in the Computing Laboratory which may require the use of MCL is a study of message distribution on a series of capability machines [Mancini1986a]. In addition, members of the COSMOS research project at Lancaster University intend to use the primitives as part of their research into distributed systems [Nicol1986a, Shrivastava1986b].

Several directions for future research into multicast communications which the author believes are worthy of consideration include:

a) the development of additional multicast applications and multicast protocols, and

b) the development of computer architectures which allow message distribution *within* a machine.

Although many different multicast applications have been discussed and implemented in the course of this thesis, there are obviously other applications which could also benefit from the use of multicast communications. This applies equally to multicast protocols, since few have been described in the literature, possibly because many distributed systems do not actively support multicast communication facilities. The primitives discussed in this thesis are intended to help overcome both of these anomalies.

As it now stands, "true" One-Group multicast transmission can only occur at the network level (as demonstrated by the Ethernet), but the distribution of messages within the machine requires a series of One-Unique transmissions. Two interesting projects could possibly be developed from this:

a) implement a shared socket facility in UNIX to allow multiple, unrelated processes to have access to a single message (i.e. allow the Processes to become members of a "true" Process group -- something UNIX does not presently support other than through implementations such as MCL). To reduce the amount of message handling required by the distribution facility, sockets could be implemented with the "copy-on-write" semantics described for the Accent Network Operating System [Rashid1985a]. By adding this feature, not only would sockets offer One-Group multicast reception within the Host, but the amount of unnecessary message copying would be reduced;

b) implement, within a machine's hardware, the ability to perform true One-Group interprocess communications on a single machine in order to reduce the overheads associated with multicast message distribution. This project, probably implementable in VLSI, would reduce or possibly even eliminate the distribution overheads discussed in Chapter Five.

References

Aguilar1984a.

L. Aguilar, "Datagram Routing for Internet Multicasting," Computer Communications Review (SIGCOMM '84), vol. 14, no. 2, pp. 58-63, June 1984.

Amamad1985a.

M. Amamad and A.J. Bernstein, "Multicast Communications in UNIX 4.2BSD," 5th International Conference on Distributed Systems, pp. 80-87, May 1985.

Banerjee1985a.

R. Banerjee and W.D. Shepherd, "The Cambridge Ring," in Local Area Networks: An Advanced Course, pp. 64-86, Springer-Verlag, Lecture Notes in Computer Science, vol. 184, 1985.

Bennett1984a.

K.H. Bennett, "Mechanisms for Distributed Control," in Distributed Computing, ed. F.B. Chambers, D.A. Duce and G.P. Jones, pp. 179-192, APIC Studies in Data Processing, no. 20, 1984.

Berglund1985a

E.J. Berglund and D.R. Cheriton, "Amaze: A Multiplayer Computer Game," IEEE Software, vol. 2, no. 3, May 1985.

Bochman1985a.

G.v. Bochman, "Specification in Distributed Systems," in Local Area Networks: An Advanced Course, pp. 470-497, Springer-Verlag, Lecture Notes in Computer Science, vol. 184, 1985.

Boggs1983a.

D.R. Boggs, "Internet Broadcasting," CSL-83-3, Xerox PARC, Palo Alto, October 1983.

Chang1984a.

J-M. Chang and N.F. Maxemchuk, "Reliable Broadcast Protocols," ACM TOCS, vol. 2, no. 3, pp. 251-273, August 1984.

Cheriton1983a.

D.R. Cheriton, "Local Networking and Internetworking in the V-System," Computer Communications Review (SIGCOMM '83), vol. 13, no. 4, pp. 9-16, October 1983.

Cheriton1984a.

D.R. Cheriton and W. Zwaenepoel, "One-to-Many Interprocess Communication in the V-System," Computer Communications Review (SIGCOMM '84), vol. 14, no. 2, p. 64, June 1984. (complete text in Stanford University report STAN-CS-84-1011).

Cheriton1984b.

D.R. Cheriton "An Experiment using Registers for Fast Message-Based Interprocess Communication," Operating Systems Review, vol. 18, no. 4, pp. 12-20, October 1984.

Cheriton1985a.

D.R. Cheriton and W. Zwaenepoel, "Distributed Process Groups in the V Kernel," ACM TOCS, vol. 3, no. 2, pp. 77-107, May 1985.

Cheriton1985b.

D.R. Cheriton and S.E. Deering, "Host Groups: A Multicast Extension for Datagram Networks," Report STAN-CS-85-1058, Department of Computer Science, Stanford University, July 1985.

Chorafas1984a.

D.N. Chorafas, Designing and Implementing Local Area Networks, McGraw-Hill, 1984.

Cohen1983a.

D. Cohen and J. Postel, "The ISO Reference Model and Other Protocol Architectures", ISI/RS-83-6, Information Sciences Institute, University of Souther California, November 1983.

DEUNA1983a.
"DEUNA User's Guide," EK-DEUNA-UG-001, Digital Equipment Corporation, 1983.

DIX1980a.
The Ethernet, A Local Area Network, Data Link Layer and Physical Layer Specifications, Digital Equipment Corporation, Intel Corporations and Xerox Corporation, Version 1.0, September 1980.

Frank1985a.
A.J. Frank, L.D. White and A.J. Bernstein, "Multicast Communications on Network Computers," IEEE Software, vol. 2, no. 3, May 1985.

Gopal1984a.
I.S. Gopal and J.M. Jaffee, "Point-to-Multipoint Communication over Broadcast Links," IEEE Transactions on Communications, vol. CON-32, no. 9, pp. 1034-1044, September 1984.

Hughes1986a.
L. Hughes, "A Multicast Communication Facility for UNIX," Computing Laboratory, University of Newcastle upon Tyne, October 1986.

Hopper1986a.
A. Hopper, S. Temple and R. Williamson, Local Area Network Design, Addison-Wesley, 1986.

IEEE1982a.
CSMA/CD Access Method and Physical Layer Specifications, IEEE Project 802, Local Area Network Standards (IEEE Computer Society), December 1982.

ISO1981a.
"Data Processing - Open Systems Interconnection - Basic Reference Model," Computer Communications Review, vol. 11, no. 2, April 1981 (ISO Draft Proposal 7498).

Janson1983a.

P.A. Janson and E. Mumprecht, "Addressing and Routing in a Hierarchy of Token Rings," in Ring Technology and Local Area Networks, ed. I.N. Dallas and E.B. Spratt, pp. 97-109, IFIP North-Holland, September 1983.

JNTIS1982a.

Cambridge Ring 82 Interface Specifications, Joint Network Team of the Computer Board and Research Councils, September 1982.

JNTPS1982a.

Cambridge Ring 82 Protocol Specifications, Joint Network Team of the Computer Board and Research Councils, November 1982.

Lakin1982a.

W.L. Lakin, The ASWE Serial Highway: A Fault Tolerant Communications System, ASWE, Portsmouth, 1982.

Lee1983a.

E.S. Lee and P.I.P. Boulton, "The Principles and Performance of Hubnet," IEEE Journal on Selected Areas in Communications, no. SAC-1(5), pp.711-720, 1983.

Leffler1982a.

S.J. Leffler, R.S. Fabry and W.N. Joy, A 4.2BSD Interprocess Communication Primer (draft), Computer Systems Research Group, University of California, Berkeley, July 1983.

Leslie1984a.

I.M. Leslie, R.M. Needham, J.W. Burren and G.C. Adams, "The Architecture of the UNIVERSE Project," Computer Communications Review (SIGCOMM '84), vol. 14, no. 2, p. 52-57, June 1984.

Linton1986a.

A. Linton, Computing Laboratory, University of Newcastle upon Tyne, Personal Communication, February 1986.

Manchini1986a.

L. Machini, Computing Laboratory, University of Newcastle upon Tyne, Personal Communication, September 1986.

Metcalfe1975a.

R.M. Metcalfe and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," CSL-75-7, Xerox PARC, Palo Alto, November 1985.

Needham1982a.

R.M. Needham and A.J. Herbert, The Cambridge Distributed Computing System, Addison-Wesley, 1982.

Nicol1986a.

J.R. Nicol, Department of Computer Science, University of Lancaster, Personal Letter, October 1986.

Padlipsky1985a.

M.A. Padlipsky, The Elements of Networking Style, Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

Panzieri1985a.

F. Panzieri, "Communications Protocols," in Local Area Networks: An Advanced Course, pp. 162-220, Springer-Verlag, Lecture Notes in Computer Science, vol. 184, 1985.

Parrington1986a.

G. Parrington, Computing Laboratory, University of Newcastle upon Tyne, Personal Communication.

Rashid1985a.

R.F. Rashid, "Network Operating Systems," in Local Area Networks: An Advanced Course, pp. 314-360, Springer-Verlag, Lecture Notes in Computer Science, vol. 184, 1985.

Shepherd1985a.

W.D. Shepherd, "LAN Internetworking," in Local Area Networks: An Advanced Course, pp. 396-427, Springer-Verlag, Lecture Notes in Computer Science, vol. 184, 1985.

Shoch1978a.

J.F. Shoch, "Internetwork Naming, Addressing, and Routing," 17th IEEE Computer Society International Conference (COMPCON), September 1978.

Shrivastava1986a.

S.K. Shrivastava, "ARJUNA Progress Report," SRM/441 (ARJ/000), Computing Laboratory, University of Newcastle upon Tyne, August 1986.

Shrivastava1986b.

S.K. Shrivastava, Computing Laboratory, University of Newcastle upon Tyne, Personal Communication, September 1986.

Tokuda1983a.

H. Tokuda and E.G. Manning, "An Interprocess Communications Model for a Distributed Software Testbed," Computer Communications Review (SIGCOMM '83), vol. 13, no. 2, pp. 205-212, March 1983.

Tsay1983a.

D.P. Tsay and M.T. Liu, "MIKE: A Network Operating System for the Distributed Double-Loop Computer Network," IEEE Transactions on Software Engineering, vol. SE-9, pp. 143-154, March 1983.

UNIX1983a.

UNIX 4.2BSD Programmers Manual, University of California, Berkeley, 1983.

Waters1984a.

A.G. Waters, C.J. Adams, I.M. Leslie, and R.M. Needham, "The Use of Broadcast Techniques on the UNIVERSE Network," Computer Communications Review (SIGCOMM '84), vol. 14, no. 2, pp. 52-57, June 1984.

Watson1983a.
R.W. Watson, "Identifiers (naming) in distributed systems," in Distributed Systems: Architecture and Implementation, pp. 191-210, Springer-Verlag, Lecture Notes in Computer Science, vol. 105, 1983.

Watson1983b.
R.W. Watson, "Distributed System Architecture Model," in Distributed Systems: Architecture and Implementation, pp. 10-56, Springer-Verlag, Lecture Notes in Computer Science, vol. 105, 1983.

Zwaenepoel1985a.
W. Zwaenepoel, "Message Passing on a Local Network," Report No. STAN-CS-85-1083, Department of Computer Science, Stanford University, October 1985.